Table of Contents

使用Spring Cloud与Docker实战微服务

本文是对Spring Cloud和Docker的实战与总结，目前在 `Git@OSC` 以及 `Github` 上同步更新。

阅读地址：虽然可直接在Git@OSC/GitHub上直接阅读。但是建议使用GitBook，阅读体验更好。

- 前往以下地址阅读：http://book.itmuch.com
- 直接访问Gitbook官网：https://eacdy.gitbooks.io/spring-cloud-book/content/
- 将代码clone到本地后，使用Typora或Atom等Markdown阅读软件进行阅读，也可使用**Gitbook**进行构建。

愿景

- Spring Cloud核心组件的系统讲解
- 帮助大家快速上手，理解Spring Cloud各组件的用途
- Docker入门
- 系统了解微服务
- 迭代计划

> 1. 如何使用Docker部署Spring Cloud应用。
> 2. 持续集成
> 3. 自动运维
> 4. 开箱可用的脚手架项目汇总
> 5. Spring Cloud最佳实践总结

转载协议

个人公众号	社区公众号

答疑&勘误&建议&意见

- http://www.spring4all.com
- 联系QQ：511932633

Spring For All社区QQ群

- ①群： 365234583（已满 1000人群）
- ②群： 123013854（已满 1000人群）
- ③群： 290714704（未满 500人群）
- ④群： 112133511（未满 500人群）
- ⑤群： 157525002（未满 2000人群）
- ⑥群： 564840207（未满 2000人群）
- ⑦群： 470962790（已满 2000人群）
- ⑧群： 613456104（已满 500人群）
- ⑨群： 534583667（未满 1000人群）
- ⑩群： 210742970（未满 500人群）

地址

本书地址：

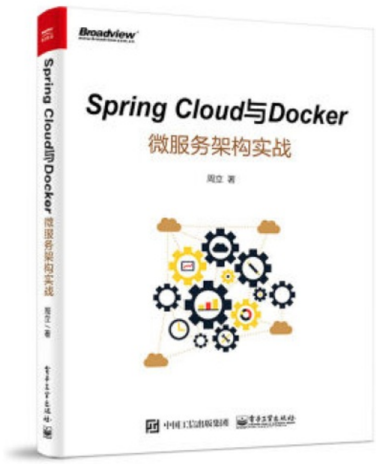
- Git@OSC地址：http://git.oschina.net/itmuch/spring-cloud-book
- Github地址：https://github.com/eacdy/spring-cloud-book

配套代码地址：

- Git@OSC地址：http://git.oschina.net/itmuch/spring-cloud-study
- Github地址：https://github.com/eacdy/spring-cloud-study

实体书

- 实体书《Spring Cloud与Docker微服务架构实战》上市啦！实体书基于Spring Cloud Camden SR4&Docker 1.13编写！京东购买链接：



Spring Cloud与Docker微服务架构实战

¥57.8

扫码购买

鸣谢

捐助名单	捐助金额	捐助方式
牛牛	200	微信红包
浩	10	支付宝转账
Жуковаский	30	支付宝转账

S1ahs3r、Leoops、牛牛、还有通过git@osc项目捐助的兄弟！

感谢其为项目作出的贡献！都是热心的兄弟！

1 微服务简介

什么是微服务架构

近年来，在软件开发领域关于微服务的讨论呈现出火爆的局面，有人倾向于在系统设计与开发中采用微服务方式实现软件系统的松耦合、跨部门开发，被认为是IT软件架构的未来方向，Martin Fowler也给微服务架构极高的评价；同时，反对之声也很强烈，持反对观点的人表示微服务增加了系统维护、部署的难度，导致一些功能模块或代码无法复用，同时微服务允许使用不同的语言和框架来开发各个系统模块，这又会增加系统集成与测试的难度，而且随着系统规模的日渐增长，微服务在一定程度上也会导致系统变得越来越复杂。尽管一些公司已经在生产系统中采用了微服务架构，并且取得了良好的效果；但更多公司还是处在观望的态度。

什么是微服务架构呢？简单说就是将一个完整的应用（单体应用）按照一定的拆分规则（后文讲述）拆分成多个不同的服务，每个服务都能独立地进行开发、部署、扩展。服务于服务之间通过注入RESTful api或其他方式调用。大家可以搜索到很多相关介绍和文章。本文暂不细表。在此推荐两个比较好的博客：

> http://microservices.io/ http://martinfowler.com/articles/microservices.html

2 Spring Cloud

Spring Cloud 简介

Spring Cloud是在Spring Boot的基础上构建的，用于简化分布式系统构建的工具集，为开发人员提供快速建立分布式系统中的一些常见的模式。

> 例如：配置管理（configuration management），服务发现（service discovery），断路器（circuit breakers），智能路由（ intelligent routing），微代理（micro-proxy），控制总线（control bus），一次性令牌（ one-time tokens），全局锁（global locks），领导选举（leadership election），分布式会话（distributed sessions），集群状态（cluster state）。

Spring Cloud 包含了多个子项目：

> 例如：Spring Cloud Config、Spring Cloud Netflix等

Spring Cloud 项目主页：http://projects.spring.io/spring-cloud/

Talk is cheap, show me the code.下面我们将以代码与讲解结合的方式，为大家讲解Spring Cloud中的各种组件。

准备工作

- 技术储备：

所需技能	备注
Java	
Maven	文章涉及到大量的代码，均使用Maven构建
Spring Boot	Spring Cloud是在Spring Boot基础上构建的

- 环境准备：

工具	版本或描述
JDK	1.8
IDE	STS 或者 IntelliJ IDEA，本教程使用的是STS.
Maven	3.x

- 本课程所使用的软件及版本：

使用到的软件	版本号	是否最新版本
Spring Boot	1.4.0.RELEASE	是
Spring Cloud	Brixton.SR5	是

- Host配置：在生产环境下，我们往往会为每个应用配置一个host，使用host而非IP进行访问。为了更加贴近生产环境，以及后文Docker章节的讲解，我们配置一下Host。在Windows系统下，是 `C:\Windows\System32\drivers\etc\hosts` 文件，在Linux系统下，是 `/etc/hosts` 文件：

Host配置
127.0.0.1 discovery config-server gateway movie user feign ribbon

- 主机规划：

项目名称	端口	描述	URL
microservice-api-gateway	8050	API Gateway	详见文章
microservice-config-client	8041	配置服务的客户端	详见文章
microservice-config-server	8040	配置服务	详见文章
microservice-consumer-movie-feign	8020	Feign Demo	/feign/1
microservice-consumer-movie-feign-with-hystrix	8021	Feign Hystrix Demo	/feign/1
microservice-consumer-movie-feign-with-hystrix-stream	8022	Hystrix Dashboard Demo	/feign/1
microservice-consumer-movie-ribbon	8010	Ribbon Demo	/ribbon/1
microservice-consumer-movie-ribbon-with-hystrix	8011	Ribbon Hystrix Demo	/ribbon/1
microservice-discovery-eureka	8761	注册中心	/
microservice-hystrix-dashboard	8030	hystrix监控	/hystrix.stream
microservice-hystrix-turbine	8031	turbine	/turbine.stream
microservice-provider-user	8000	服务提供者	/1

- Spring Cloud所有的配置项：

http://cloud.spring.io/spring-cloud-static/Brixton.SR5/#_appendix_compendium_of_configuration_properties

父项目的建立

在进入主题之前，我们首先创建一个父项目（spring-cloud-microservice-study），这样可以对项目中的Maven依赖进行统一的管理。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
```

```xml
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.itmuch.cloud</groupId>
  <artifactId>spring-cloud-microservice-study</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>microservice-discovery-eureka</module>
    <module>microservice-provider-user</module>
    <module>microservice-consumer-movie-ribbon</module>
    <module>microservice-consumer-movie-feign</module>
    <module>microservice-consumer-movie-ribbon-with-hystrix</mod
ule>
    <module>microservice-consumer-movie-feign-with-hystrix</modu
le>
    <module>microservice-hystrix-dashboard</module>
    <module>microservice-consumer-movie-feign-with-hystrix-stream
</module>
    <module>microservice-hystrix-turbine</module>
    <module>microservice-config-server</module>
    <module>microservice-config-client</module>
    <module>microservice-config-server-eureka</module>
    <module>microservice-config-client-eureka</module>
    <module>microservice-api-gateway</module>
  </modules>

  <!-- 使用最新的spring-boot版本 -->
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEnc
oding>
    <java.version>1.8</java.version>
  </properties>
```

```
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Brixton.SR5</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

TIPS

1. 笔者讲解采用最新的Spring Boot 和Spring Cloud进行讲解，其中可能涉及到部分新特性，笔者尽量指出，同时笔者能力有限，如有理解不到位的地方，还请各位看客指出，定在第一时间进行修正。
2. Spring Cloud版本并不是传统的使用数字的方式标识，而是使用例如：Angel、Brixton、Camden...等等伦敦的地名来命名版本，版本的先后顺序大家可能已经猜到了，就是使用字母表的先后来标识的。笔者在咨询过Spring Cloud的主要贡献者之一Josh Long之后已经确认。

2.1 服务发现

关于服务发现

在微服务架构中，服务发现（Service Discovery）是关键原则之一。手动配置每个客户端或某种形式的约定是很难做的，并且很脆弱。Spring Cloud提供了多种服务发现的实现方式，例如：Eureka、Consul、Zookeeper。

Spring Cloud支持得最好的是Eureka，其次是Consul，最次是Zookeeper。

2.1.1 Eureka

准备工作

- 在生产环境下，我们往往会为每个应用配置一个host，使用host而非IP进行访问。为了更加贴近生产环境，以及后文Docker章节的讲解，我们首先配置一下Host

```
127.0.0.1 discovery
```

代码示例

- 创建一个Maven工程（microservice-discovery-eureka），并在pom.xml中加入如下内容：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-discovery-eureka</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka-server</artifactId>

    </dependency>
  </dependencies>
</project>
```

- 编写Spring Boot启动程序：通过@EnableEurekaServer申明一个注册中心：

```java
/**
 * 使用Eureka做服务发现。
 * @author eacdy
 */
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
  public static void main(String[] args) {
    SpringApplication.run(EurekaApplication.class, args);
  }
}
```

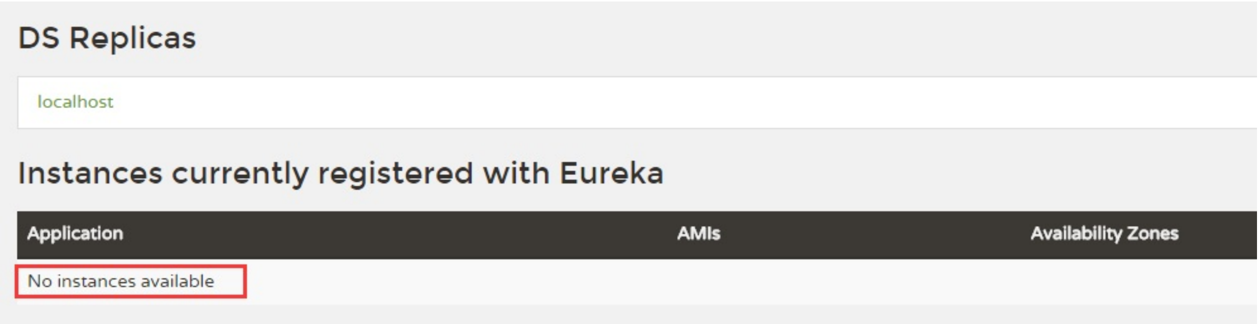
- 在默认情况下，Eureka会将自己也作为客户端尝试注册，所以在单机模式下，我们需要禁止该行为，只需要在application.yml中如下配置：

```yaml
server:
  port: 8761                    # 指定该Eureka实例的端口

eureka:
  instance:
    hostname: discovery         # 指定该Eureka实例的主机名
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.p
ort}/eureka/

# 参考文档：http://projects.spring.io/spring-cloud/docs/1.0.3/spri
ng-cloud.html#_standalone_mode
# 参考文档：http://my.oschina.net/buwei/blog/618756
```

- 启动工程后，访问：http://discovery:8761/ ，如下图。我们会发现此时还没有服务注册到Eureka上面。



代码地址（任选其一）

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-discovery-eureka https://github.com/eacdy/spring-cloud-study/tree/master/microservice-discovery-eureka

2.1.2 Eureka的高可用

按照前文对Eureka的讲解，我们即可构建出一个简单的注册中心。但此时的Eureka是单点的，不适合于生产环境，那么如何实现Eureka的高可用呢？

- 添加主机名：

```
127.0.0.1 peer1 peer2
```

- 修改application.yml

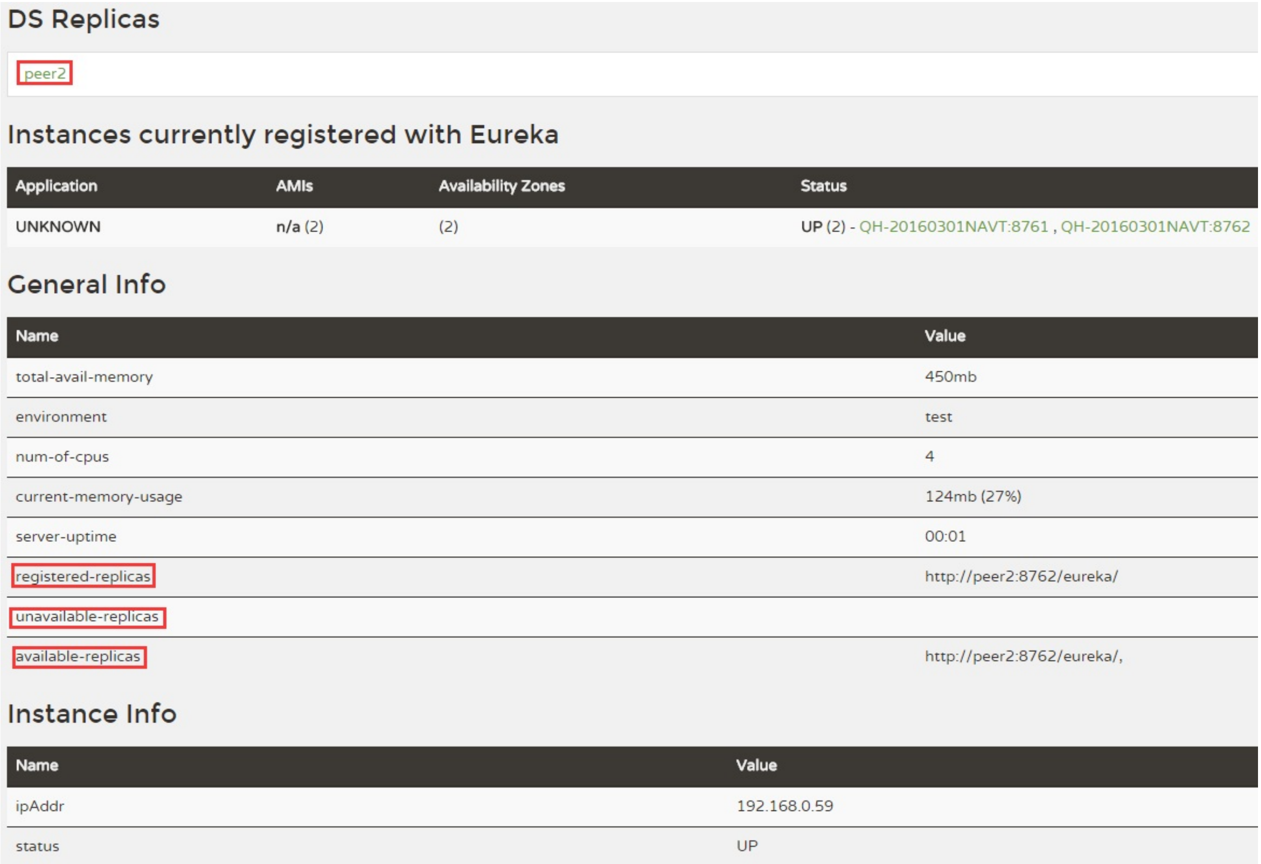
```
---
spring:
  profiles: peer1                                        # 指定profile=peer1
server:
  port: 8761
eureka:
  instance:
    hostname: peer1                                      # 指定当profile=peer1时，主机名
  client:
    serviceUrl:
      defaultZone: http://peer2:8762/eureka/             # 将自己注册到peer2这个Eureka上面去

---
spring:
  profiles: peer2
server:
  port: 8762
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/
```

- 分别启动两个Eureka应用：

```
java -jar microservice-discovery-eureka-0.0.1-SNAPSHOT.jar --spring.profiles.active=peer1
java -jar microservice-discovery-eureka-0.0.1-SNAPSHOT.jar --spring.profiles.active=peer2
```

- 访问 `http://peer1:8761` ，我们会发现 `registered-replicas` 中已经有 `peer2` 节点了，同样地，访问 `http://peer2:8762` ，也能发现其中的 `registered-replicas` 有 `peer1` 节点，如下图：



DS Replicas

peer2

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
UNKNOWN	n/a (2)	(2)	UP (2) - QH-20160301NAVT:8761 , QH-20160301NAVT:8762

General Info

Name	Value
total-avail-memory	450mb
environment	test
num-of-cpus	4
current-memory-usage	124mb (27%)
server-uptime	00:01
registered-replicas	http://peer2:8762/eureka/
unavailable-replicas	
available-replicas	http://peer2:8762/eureka/,

Instance Info

Name	Value
ipAddr	192.168.0.59
status	UP

- 我们尝试将 `peer2` 节点关闭，然后访问 `http://peer1:8761` ，会发现此时peer2会被添加到 `unavaliable-replicas` 一栏中。

注意：

> 该示例中的 `hostname` 并非必须的，如果不配置，默认将会使用IP进行查找。

将服务注册到高可用的**Eureka**

如果注册中心是高可用的，那么各个微服务配置只需要将 `defaultZone` 改为如下即可：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/,http://peer2:8762/eureka
```

参考文档

> http://cloud.spring.io/spring-cloud-static/Brixton.SR5/#spring-cloud-eureka-server

2.1.3 Consul

Consul 是 HashiCorp 公司推出的开源工具，用于实现分布式系统的服务发现与配置。与其他分布式服务注册与发现的方案，Consul的方案更“一站式”，内置了服务注册与发现框 架、分布一致性协议实现、健康检查、Key/Value存储、多数据中心方案，不再需要依赖其他工具（比如ZooKeeper等）。使用起来也较 为简单。Consul使用Go语言编写，因此具有天然可移植性(支持Linux、windows和Mac OS X)；安装包仅包含一个可执行文件，方便部署，与Docker等轻量级容器可无缝配合。

下面以Consul 在CentOS 7系统下为例，讲解Consul的安装及使用，其他平台的安装也是类似的，所使用的Consul 版本为v0.7.0。

Consul下载页面：https://www.consul.io/downloads.html 。

2.1.4 Consul安装与使用

准备工作

一台CentOS 7 机器，输入 `ifconfig` ，查看网卡信息如下：

```
eno16777736: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.11.143  netmask 255.255.255.0  broadcast 192.168.11.255
        inet6 fe80::20c:29ff:fe89:6b91  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:89:6b:91  txqueuelen 1000  (Ethernet)
        RX packets 752526  bytes 705406371 (672.7 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 142062  bytes 18646825 (17.7 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 172  bytes 1003766 (980.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 172  bytes 1003766 (980.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

我们可以看到，该机器有两个IP：

```
192.168.11.143
127.0.0.1
```

Consul的安装与启动

- 安装Consul（以CentOS7为例）：

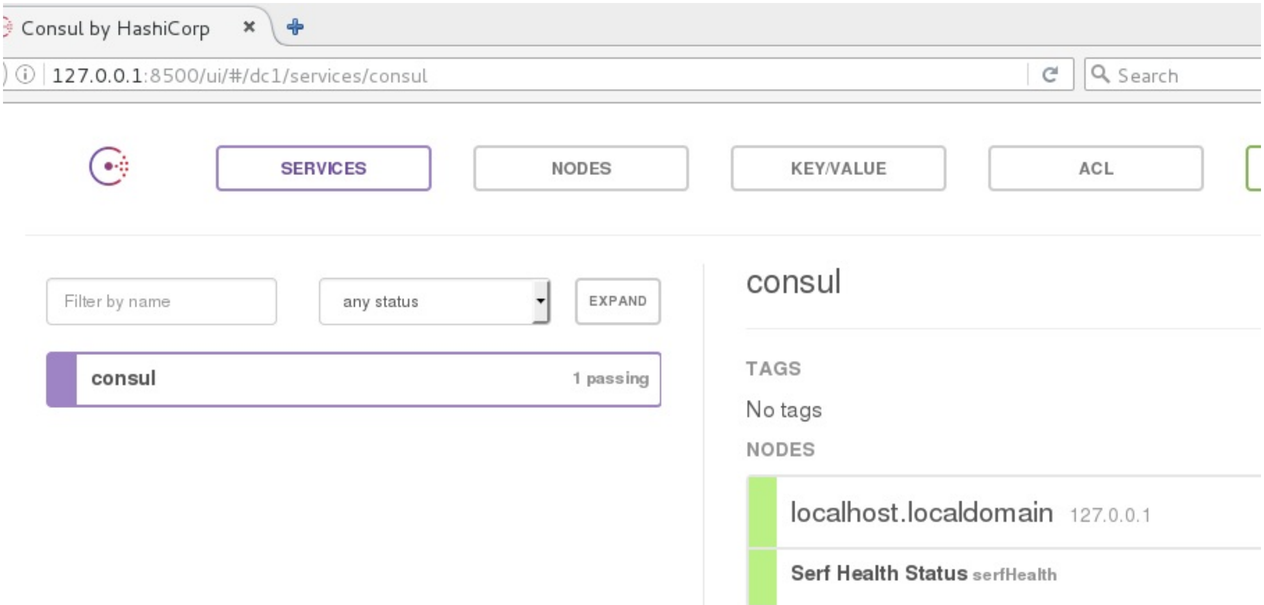
```
cd /usr/local/bin/
wget https://releases.hashicorp.com/consul/0.7.0/consul_0.7.0_li
nux_amd64.zip
unzip consul0.7.0linux_amd64.zip
```

得到 `consul` 文件，这样就完成了安装。

- 启动Consul

```
./consul agent -dev          # -dev表示开发模式运行，另外还有-serve
r表示服务模式运行
```

- 输入 `http://127.0.0.1:8500/ui/` 访问Consul，可查看到如下界面：



4. 我们尝试访问 `http://192.168.11.143/ui/` ，会发现无法访问。说明Consul还不能被远程访问。那么如何设置才能被远程访问呢？Consul如何高可用呢？

2.1.5 Consul常用命令

Consul常用命令

命令	解释	示例
agent	运行一个consul agent	consul agent -dev
join	将agent加入到consul集群	consul join IP
members	列出consul cluster集群中的members	consul members
leave	将节点移除所在集群	consul leave

`consul agent` 命令详解

输入 `consul agent --help` ，可以看到 `consul agent` 的选项，如下：

```
  -advertise=addr          Sets the advertise address to use
  -advertise-wan=addr      Sets address to advertise on wan inst
ead of advertise addr
  -atlas=org/name          Sets the Atlas infrastructure name, e
nables SCADA.
  -atlas-join              Enables auto-joining the Atlas cluste
r
  -atlas-token=token       Provides the Atlas API token
  -atlas-endpoint=1.2.3.4  The address of the endpoint for Atlas
 integration.
  -bootstrap               Sets server to bootstrap mode
  -bind=0.0.0.0            Sets the bind address for cluster com
munication
  -http-port=8500          Sets the HTTP API port to listen on
  -bootstrap-expect=0      Sets server to expect bootstrap mode.
  -client=127.0.0.1        Sets the address to bind for client a
ccess.
                           This includes RPC, DNS, HTTP and HTTP
S (if configured)
```

```
  -config-file=foo          Path to a JSON file to read configura
tion from.
                            This can be specified multiple times.
  -config-dir=foo           Path to a directory to read configura
tion files
                            from. This will read every file endin
g in ".json"
                            as configuration in this directory in
 alphabetical
                            order. This can be specified multiple
 times.
  -data-dir=path            Path to a data directory to store age
nt state
  -dev                      Starts the agent in development mode.
  -recursor=1.2.3.4         Address of an upstream DNS server.
                            Can be specified multiple times.
  -dc=east-aws              Datacenter of the agent (deprecated:
use 'datacenter' instead).
  -datacenter=east-aws      Datacenter of the agent.
  -encrypt=key              Provides the gossip encryption key
  -join=1.2.3.4             Address of an agent to join at start
time.
                            Can be specified multiple times.
  -join-wan=1.2.3.4         Address of an agent to join -wan at s
tart time.
                            Can be specified multiple times.
  -retry-join=1.2.3.4       Address of an agent to join at start
time with
                            retries enabled. Can be specified mul
tiple times.
  -retry-interval=30s       Time to wait between join attempts.
  -retry-max=0              Maximum number of join attempts. Defa
ults to 0, which
                            will retry indefinitely.
  -retry-join-wan=1.2.3.4   Address of an agent to join -wan at s
tart time with
                            retries enabled. Can be specified mul
tiple times.
  -retry-interval-wan=30s   Time to wait between join -wan attemp
ts.
```

```
  -retry-max-wan=0          Maximum number of join -wan attempts.
 Defaults to 0, which
                            will retry indefinitely.
  -log-level=info           Log level of the agent.
  -node=hostname            Name of this node. Must be unique in
the cluster
  -protocol=N               Sets the protocol version. Defaults t
o latest.
  -rejoin                   Ignores a previous leave and attempts
 to rejoin the cluster.
  -server                   Switches agent to server mode.
  -syslog                   Enables logging to syslog
  -ui                       Enables the built-in static web UI se
rver
  -ui-dir=path              Path to directory containing the Web
UI resources
  -pid-file=path            Path to file to store agent PID
```

`consul agent` 命令的常用选项，如下：

- -data-dir
    - 作用：指定agent储存状态的数据目录
    - 这是所有agent都必须的
    - 对于server尤其重要，因为他们必须持久化集群的状态
- -config-dir
    - 作用：指定service的配置文件和检查定义所在的位置
    - 通常会指定为"某一个路径/consul.d"（通常情况下，.d表示一系列配置文件存放的目录）
- -config-file
    - 作用：指定一个要装载的配置文件
    - 该选项可以配置多次，进而配置多个配置文件（后边的会合并前边的，相同的值覆盖）
- -dev
    - 作用：创建一个开发环境下的server节点
    - 该参数配置下，不会有任何持久化操作，即不会有任何数据写入到磁盘
    - 这种模式不能用于生产环境（因为第二条）
- -bootstrap-expect
    - 作用：该命令通知consul server我们现在准备加入的server节点个数，该

参数是为了延迟日志复制的启动直到我们指定数量的server节点成功的加入后启动。

- -node
  - 作用：指定节点在集群中的名称
  - 该名称在集群中必须是唯一的（默认采用机器的host）
  - 推荐：直接采用机器的IP
- -bind
  - 作用：指明节点的IP地址
  - 有时候不指定绑定IP，会报 `Failed to get advertise address: Multiple private IPs found. Please configure one.` 的异常
- -server
  - 作用：指定节点为server
  - 每个数据中心（DC）的server数推荐至少为1，至多为5
  - 所有的server都采用raft一致性算法来确保事务的一致性和线性化，事务修改了集群的状态，且集群的状态保存在每一台server上保证可用性
  - server也是与其他DC交互的门面（gateway）
- -client
  - 作用：指定节点为client，指定客户端接口的绑定地址，包括：HTTP、DNS、RPC
  - 默认是127.0.0.1，只允许回环接口访问
  - 若不指定为-server，其实就是-client
- -join
  - 作用：将节点加入到集群
- -datacenter（老版本叫-dc，-dc已经失效）
  - 作用：指定机器加入到哪一个数据中心中

如上，大家应该可以猜到，

使用 `-client` 参数可指定允许客户端使用什么ip去访问，例如 `-client 192.168.11.143` 表示可以使用 `http://192.168.11.143:8500/ui` 去访问。

我们尝试一下：

```
consul agent -dev -client 192.168.11.143
```

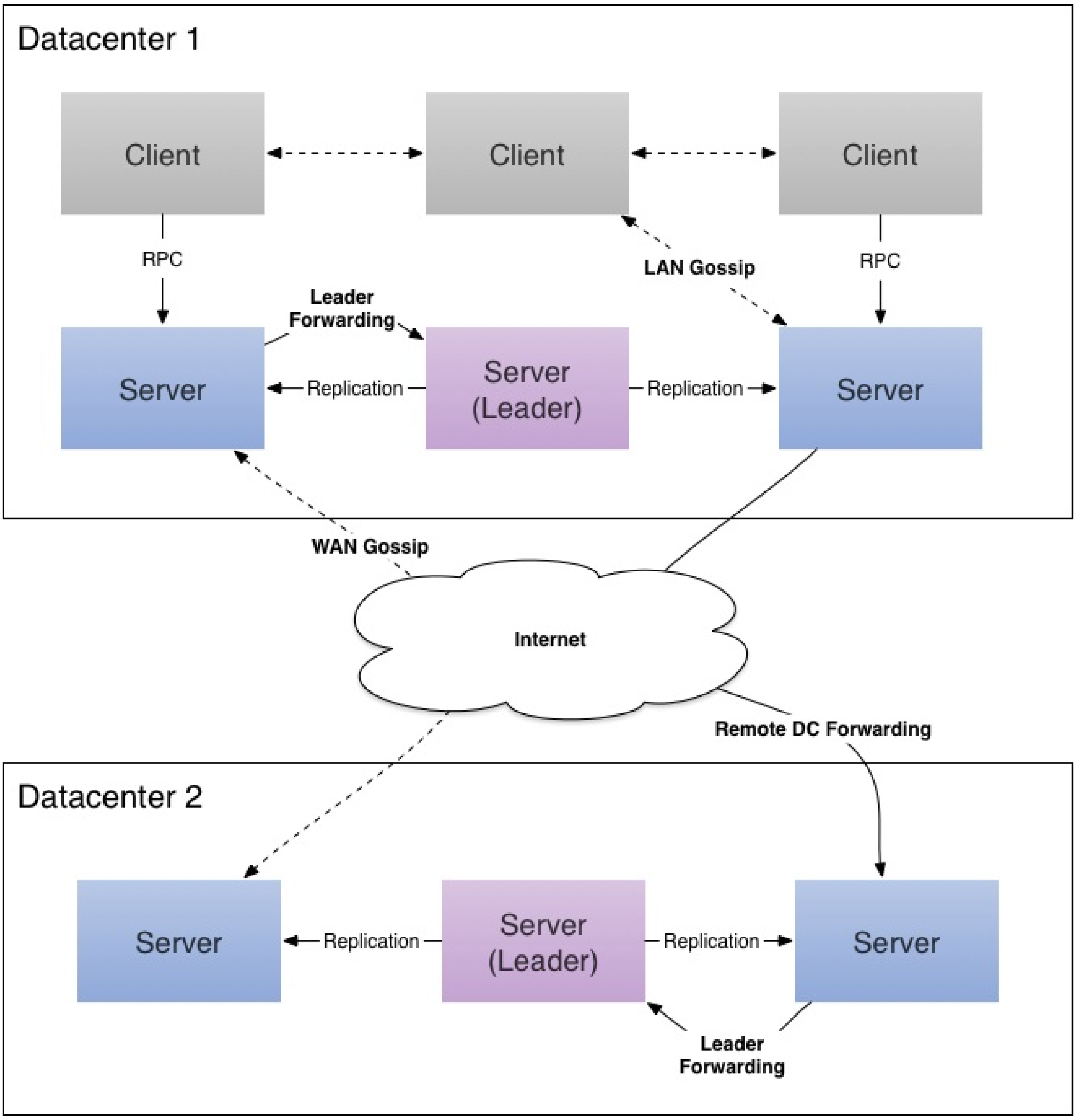
发现果然可以使用 `http://192.168.11.143:8500/ui` 访问了。

参考文档

> 官方文档：https://www.consul.io/docs/agent/options.html
>
> Consul系列博客：http://www.cnblogs.com/java-zhao/p/5378876.html

2.1.6 Consul 的高可用

Consul Cluster集群架构图如下：



这边准备了三台CentOS 7的虚拟机，主机规划如下，供参考：

主机名称	IP	作用	是否允许远程访问
node0	192.168.11.143	consul server	是
node1	192.168.11.144	consul client	否
node2	192.168.11.145	consul client	是

搭建步骤：

- 启动node0机器上的Consul（node0机器上执行）：

```
consul agent -data-dir /tmp/node0 -node=node0 -bind=192.168.11.143 -datacenter=dc1 -ui -client=192.168.11.143 -server -bootstrap-expect 1
```

- 启动node1机器上的Consul（node1机器上执行）：

```
consul agent -data-dir /tmp/node1 -node=node1 -bind=192.168.11.144 -datacenter=dc1 -ui
```

- 启动node2机器上的Consul（node2机器上执行）：

```
consul agent -data-dir /tmp/node2 -node=node2 -bind=192.168.11.145 -datacenter=dc1 -ui -client=192.168.11.145
```

- 将node1节点加入到node0上（node1机器上执行）：

```
consul join 192.168.11.143
```

- 将node2节点加入到node0上（node2机器上执行）：

```
consul join -rpc-addr=192.168.11.145:8400  192.168.11.143
```

- 这样一个简单的Consul集群就搭建完成了，在node1上查看当前集群节点：

```
consul members -rpc-addr=192.168.11.143:8400
```

结果如下：

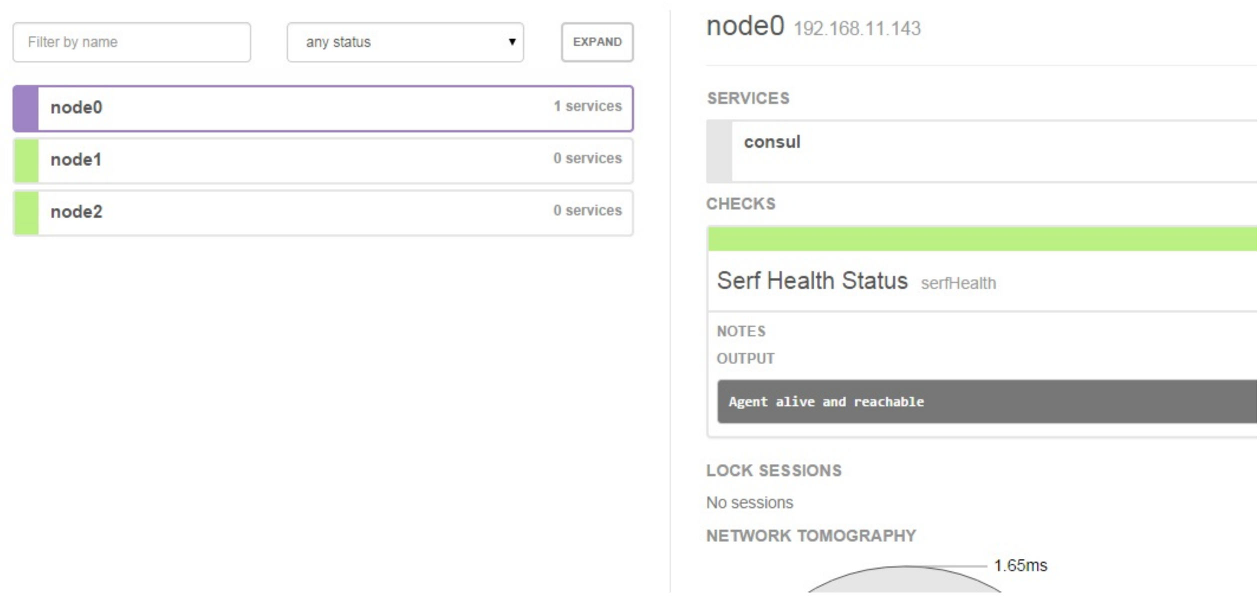
```
Node   Address              Status  Type    Build  Protocol  DC
node0  192.168.11.143:8301  alive   server  0.7.0  2         dc1
node1  192.168.11.144:8301  alive   client  0.7.0  2         dc1
node2  192.168.11.145:8301  alive   client  0.7.0  2         dc1
```

说明集群已经搭建成功了。

我们分析一下，为什么第5步和第6步需要加 `-rpc-addr` 选项，而第4步不需要加任何选项呢？原因是 `-client` 指定了客户端接口的绑定地址，包括：HTTP、DNS、RPC，而 `consul join` 、 `consul members` 都是通过RPC与Consul交互的。

访问集群

如上，我们三个节点都加了 `-ui` 参数启动了内建的界面。我们可以通过： `http://192.168.11.143:8500/ui/` 或者 `http://192.168.11.145:8500/ui/` 进行访问，也可以在node1机器上通过 `http://127.0.0.1:8500/ui/` 进行访问，原因是node1没有开启远程访问 ，三种访问方式结果是一致的，如下：



参考文档：

> Consul官方文档：https://www.consul.io/intro/getting-started/install.html
>
> Consul 系列博文：http://www.cnblogs.com/java-zhao/archive/2016/04/13/5387105.html
>
> 使用consul实现分布式服务注册和发现：http://www.tuicool.com/articles/M3QFven

2.2 服务提供者

首先说明一下，为了便于讲解，本节之后，如无特殊说明，均是以单点的Eureka进行讲解的。

服务提供者和服务消费者

下面这张表格，简单描述了服务提供者/消费者是什么：

名词	概念
服务提供者	服务的被调用方（即：为其他服务提供服务的服务）
服务消费者	服务的调用方（即：依赖其他服务的服务）

服务提供者代码示例

这是一个稍微有点复杂的程序。我们使用spring-data-jpa操作h2数据库，同时将该服务注册到注册中心Eureka中。

- 创建一个Maven工程，并在pom.xml中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-provider-user</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
```

```
  <dependencies>
    <!-- 添加Eureka的依赖 -->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

- 配置文件：application.yml

```
server:
  port: 8000
spring:
  application:
    name: microservice-provider-user    # 项目名称尽量用小写
  jpa:
    generate-ddl: false
    show-sql: true
    hibernate:
      ddl-auto: none
  datasource:                           # 指定数据源
    platform: h2                        # 指定数据源类型
    schema: classpath:schema.sql        # 指定h2数据库的建表脚本
    data: classpath:data.sql            # 指定h2数据库的insert脚本
logging:
  level:
    root: INFO
    org.hibernate: INFO
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE
    org.hibernate.type.descriptor.sql.BasicExtractor: TRACE
    com.itmuch.youran.persistence: ERROR
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/    # 指定注册中心
的地址
  instance:
    preferIpAddress: true
```

- 建表语句：schema.sql

```
drop table user if exists;
create table user (id bigint generated by default as identity, u
sername varchar(255), age int, primary key (id));
```

- 插库语句：data.sql

```
insert into user (id, username, age) values (1,'Tom',12);
insert into user (id, username, age) values (2,'Jerry', 23);
insert into user (id, username, age) values (3,'Reno', 44);
insert into user (id, username, age) values (4,'Josh', 55);
```

- DAO：

```
@Repository
public interface UserRepository extends JpaRepository<User, Long
> {
}
```

- Controller：

```
/**
 * 作用：
 * ① 测试服务实例的相关内容
 * ② 为后来的服务做提供
 * @author eacdy
 */
@RestController
public class UserController {
  @Autowired
  private DiscoveryClient discoveryClient;
  @Autowired
  private UserRepository userRepository;

  /**
   * 注：@GetMapping("/{id}")是spring 4.3的新注解等价于：
   * @RequestMapping(value = "/id", method = RequestMethod.GET)
   * 类似的注解还有@PostMapping等等
   * @param id
   * @return user信息
   */
  @GetMapping("/{id}")
  public User findById(@PathVariable Long id) {
    User findOne = this.userRepository.findOne(id);
    return findOne;
  }

  /**
   * 本地服务实例的信息
   * @return
   */
  @GetMapping("/instance-info")
  public ServiceInstance showInfo() {
    ServiceInstance localServiceInstance = this.discoveryClient.
getLocalServiceInstance();
    return localServiceInstance;
  }
}
```

- 实体类：

```
@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  @Column
  private String username;
  @Column
  private Integer age;
  ...
  // getters and setters
}
```

- 编写Spring Boot启动程序，通过@EnableDiscoveryClient注解，即可将microservice-provider-user服务注册到Eureka上面去

```
@SpringBootApplication
@EnableDiscoveryClient
public class UserApplication {
  public static void main(String[] args) {
    SpringApplication.run(UserApplication.class, args);
  }
}
```

至此，代码编写完成。

测试

我们依次启动Eureka服务和microservice-provider-user服务。

访问：http://localhost:8761，如下图。我们会发现microservice-provider-user服务已经被注册到了Eureka上面了。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - QH-20160301NAVT:microservice-provider-user:8000

访问：http://localhost:8000/instance-info，返回结果：

```
{
    "host": "192.168.0.59",
    "port": 8000,
    "metadata": {},
    "uri": "http://192.168.0.59:8000",
    "secure": false,
    "serviceId": "microservice-provider-user"
}
```

访问：http://discovery:8000/1，返回结果：

```
{
    "id": 1,
    "username": "Tom",
    "age": 12
}
```

服务注册到高可用Eureka

如果 `Eureka` 是高可用的，那么各个微服务配置只需要将 `defaultZone` 改为如下即可：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/,http://peer2:8762/e
ureka
```

代码地址（任选其一）

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-provider-user https://github.com/eacdy/spring-cloud-study/tree/master/microservice-provider-user

2.3 服务消费者

上文我们创建了注册中心，以及服务的提供者microservice-provider-user，并成功地将服务提供者注册到了注册中心上。

要想消费microservice-provider-user的服务是很简单的，我们只需要使用RestTemplate即可，或者例如HttpClient之类的http工具也是可以的。但是在集群环境下，我们必然是每个服务部署多个实例，那么服务消费者消费服务提供者时的负载均衡又要如何做呢？

准备工作

1. 启动注册中心：microservice-discovery-eureka
2. 启动服务提供方：microservice-provider-user
3. 修改microservice-provider-user的端口为8001，另外启动一个实例

此时，访问http://discovery:8761

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (2)	(2)	UP (2) - QH-20160301NAVT:microservice-provider-user:8001 , QH-20160301NAVT:microservice-provider-user:8000

可以在Eureka中看到microservice-provider-user有两个实例在运行。

下面我们创建一个新的微服务（microservice-consumer-movie-*），负载均衡地消费microservice-provider-user的服务。

2.3.1 Ribbon

Ribbon介绍

Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法，将Netflix的中间层服务连接在一起。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。简单的说，就是在配置文件中列出Load Balancer后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。简单地说，Ribbon是一个客户端负载均衡器。

Ribbon工作时分为两步：第一步先选择 Eureka Server, 它优先选择在同一个Zone且负载较少的Server；第二步再根据用户指定的策略，在从Server取到的服务注册列表中选择一个地址。其中Ribbon提供了多种策略，例如轮询、随机、根据响应时间加权等。

Ribbon代码示例

创建一个Maven项目，并在pom.xml中加入如下内容：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-consumer-movie-ribbon</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>

    <!-- 整合ribbon -->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-ribbon</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类：MovieRibbonApplication.java。使用@LoadBalanced注解，为RestTemplate开启负载均衡的能力。

```
@SpringBootApplication
@EnableDiscoveryClient
public class MovieRibbonApplication {
  /**
   * 实例化RestTemplate，通过@LoadBalanced注解开启均衡负载能力.
   * @return restTemplate
   */
  @Bean
  @LoadBalanced
  public RestTemplate restTemplate() {
    return new RestTemplate();
  }

  public static void main(String[] args) {
    SpringApplication.run(MovieRibbonApplication.class, args);
  }
}
```

实体类：User.java

```
public class User {
  private Long id;
  private String username;
  private Integer age;
  ...
  // getters and setters
}
```

Ribbon的测试类：RibbonService.java

```java
@Service
public class RibbonService {
  @Autowired
  private RestTemplate restTemplate;

  public User findById(Long id) {
    // http://服务提供者的serviceId/url
    return this.restTemplate.getForObject("http://microservice-p
rovider-user/" + id, User.class);
  }
}
```

controller：RibbonController.java

```java
@RestController
public class RibbonController {
  @Autowired
  private RibbonService ribbonService;

  @GetMapping("/ribbon/{id}")
  public User findById(@PathVariable Long id) {
    return this.ribbonService.findById(id);
  }
}
```

application.yml

```
server:
  port: 8010
spring:
  application:
    name: microservice-consumer-movie-ribbon
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    preferIpAddress: true
```

启动后，访问多次http://localhost:8010/ribbon/1，返回结果：

```
{
    "id": 1,
    "username": "Tom",
    "age": 12
}
```

然后打开两个microservice-provider-user实例的控制台，发现两个实例都输出了类似如下的日志内容：

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_
, user0_.username as username3_0_0_ from user user0_ where user0
_.id=?
2016-09-13 21:38:56.719 TRACE 17404 --- [nio-8000-exec-1] o.h.ty
pe.descriptor.sql.BasicBinder      : binding parameter [1] as [B
IGINT] - [1]
2016-09-13 21:38:56.720 TRACE 17404 --- [nio-8000-exec-1] o.h.ty
pe.descriptor.sql.BasicExtractor   : extracted value ([age2_0_0_
] : [INTEGER]) - [12]
2016-09-13 21:38:56.720 TRACE 17404 --- [nio-8000-exec-1] o.h.ty
pe.descriptor.sql.BasicExtractor   : extracted value ([username3
_0_0_] : [VARCHAR]) - [Tom]
2016-09-13 21:39:10.588  INFO 17404 --- [trap-executor-0] c.n.d.
s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints
via configuration
```

至此，我们已经通过Ribbon在客户端侧实现了均衡负载。

代码地址（任选其一）：

Ribbon代码地址：

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon

2.3.2. Feign

Feign介绍

Feign是一个声明式的web service客户端，它使得编写web service客户端更为容易。创建接口，为接口添加注解，即可使用Feign。Feign可以使用Feign注解或者JAX-RS注解，还支持热插拔的编码器和解码器。Spring Cloud为Feign添加了Spring MVC的注解支持，并整合了Ribbon和Eureka来为使用Feign时提供负载均衡。

> 译自：http://projects.spring.io/spring-cloud/docs/1.0.3/spring-cloud.html#spring-cloud-feign

Feign示例

创建一个Maven项目，并在pom.xml添加如下内容：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-consumer-movie-feign</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-feign</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类：MovieFeignApplication.java

```
/**
 * 使用@EnableFeignClients开启Feign
 * @author eacdy
 */
@SpringBootApplication
@EnableFeignClients
@EnableDiscoveryClient
public class MovieFeignApplication {
  public static void main(String[] args) {
    SpringApplication.run(MovieFeignApplication.class, args);
  }
}
```

实体类：User.java

```
public class User {
  private Long id;
  private String username;
  private Integer age;
  ...
  // getters and setters
}
```

Feign测试类：UserFeignClient.java。

```
/**
 * 使用@FeignClient("microservice-provider-user")注解绑定microservice-provider-user服务，还可以使用url参数指定一个URL。
 * @author eacdy
 */
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
  @RequestMapping("/{id}")
  public User findByIdFeign(@RequestParam("id") Long id);
}
```

Feign的测试类：FeignController.java

```
@RestController
public class FeignController {
  @Autowired
  private UserFeignClient userFeignClient;

  @GetMapping("feign/{id}")
  public User findByIdFeign(@PathVariable Long id) {
    User user = this.userFeignClient.findByIdFeign(id);
    return user;
  }
}
```

application.yml

```
server:
  port: 8020
spring:
  application:
    name: microservice-consumer-movie-feign
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    preferIpAddress: true
ribbon:
  eureka:
    enabled: true         # 默认为true。如果设置为false，Ribbon将不会从Eureka中获得服务列表，而是使用静态配置的服务列表。静态服务列表可使用：<client>.ribbon.listOfServers来指定。参考：http://projects.spring.io/spring-cloud/docs/1.0.3/spring-cloud.html#spring-cloud-ribbon-without-eureka

### 参考：https://spring.io/guides/gs/client-side-load-balancing/
```

同样的，启动该应用，多次访问http://localhost:8020/feign/1，我们会发现和Ribbon示例一样实现了负载均衡。

代码地址（任选其一）：

Feign代码地址：

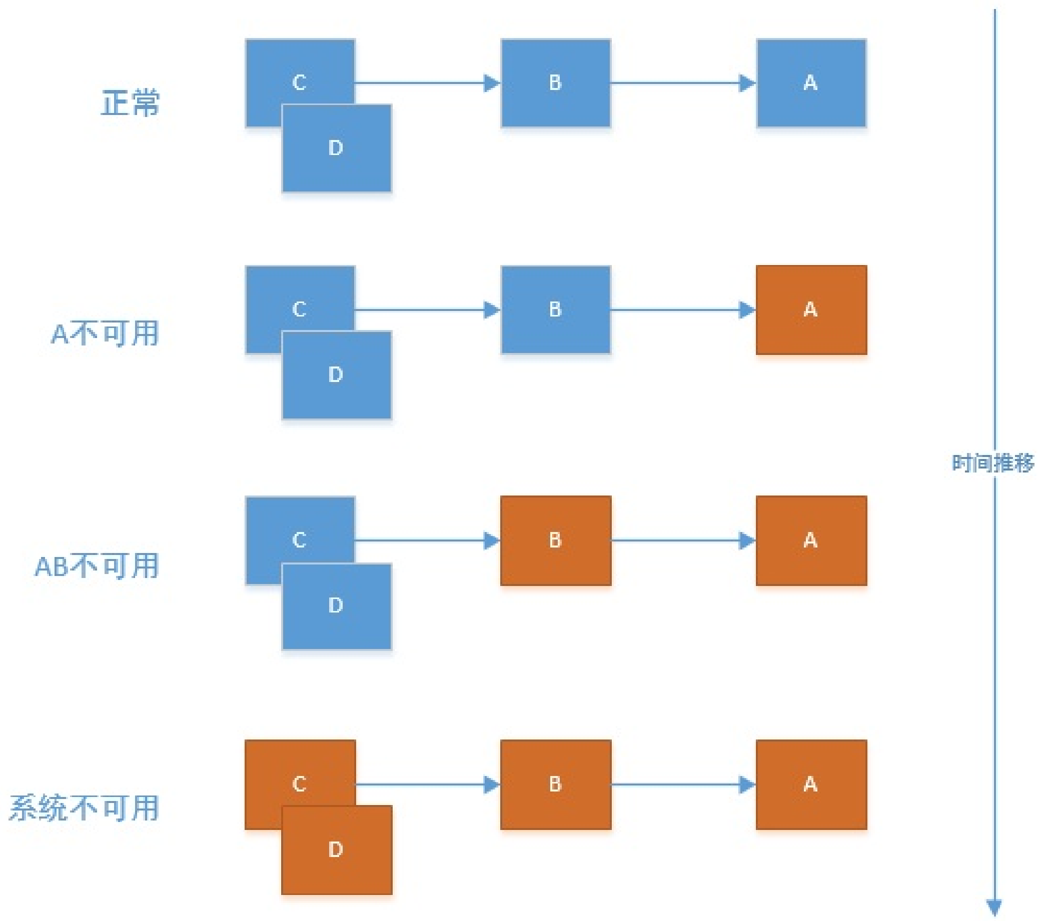
> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-feign https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-feign

2.4 熔断器

雪崩效应

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

如果下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。

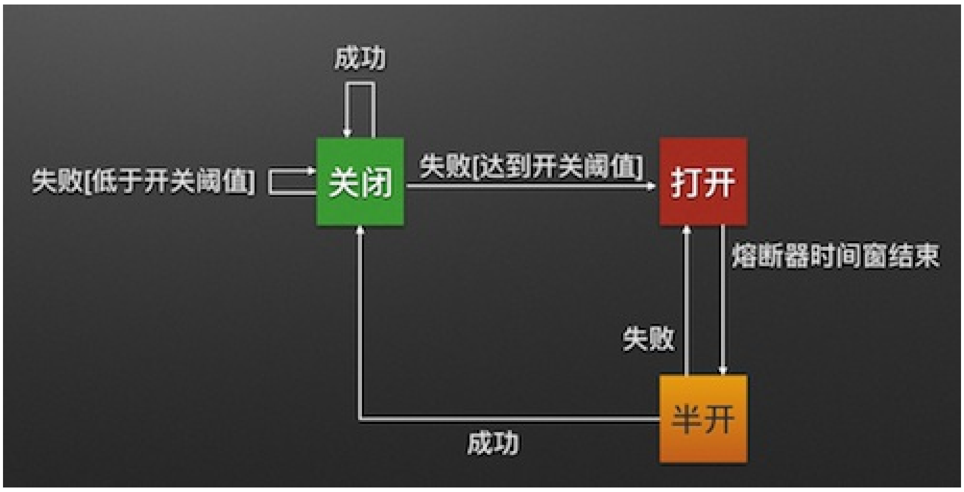


熔断器（CircuitBreaker）

熔断器的原理很简单，如同电力过载保护器。它可以实现快速失败，如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。

熔断器模式就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调用发生错误的次数，然后决定使用允许操作继续，或者立即返回错误。

熔断器开关相互转换的逻辑如下图：



2.4.1. Hystrix

Hystrix

在Spring Cloud中使用了Netflix开发的Hystrix来实现熔断器。下面我们依然通过几个简单的代码示例，进入Hystrix的学习：

通用方式使用Hystrix

代码示例：

新建一个Maven项目，在pom.xml中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-consumer-movie-ribbon-with-hystrix</a
rtifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
```

```
    <!-- 整合ribbon -->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-ribbon</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <!-- 整合hystrix -->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类：MovieRibbonHystrixApplication.java，使用@EnableCircuitBreaker注解开启断路器功能：

```
/**
 * 使用@EnableCircuitBreaker注解开启断路器功能
 * @author eacdy
 */
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class MovieRibbonHystrixApplication {
  /**
   * 实例化RestTemplate，通过@LoadBalanced注解开启均衡负载能力.
   * @return restTemplate
   */
  @Bean
  @LoadBalanced
  public RestTemplate restTemplate() {
    return new RestTemplate();
  }

  public static void main(String[] args) {
    SpringApplication.run(MovieRibbonHystrixApplication.class, a
rgs);
  }
}
```

实体类：User.java

```
public class User {
  private Long id;
  private String username;
  private Integer age;
  ...
  // getters and setters
}
```

Hystrix业务类：RibbonHystrixService.java，使用@HystrixCommand注解指定当该方法发生异常时调用的方法

```java
@Service
public class RibbonHystrixService {
  @Autowired
  private RestTemplate restTemplate;
  private static final Logger LOGGER = LoggerFactory.getLogger(RibbonHystrixService.class);

  /**
   * 使用@HystrixCommand注解指定当该方法发生异常时调用的方法
   * @param id id
   * @return 通过id查询到的用户
   */
  @HystrixCommand(fallbackMethod = "fallback")
  public User findById(Long id) {
    return this.restTemplate.getForObject("http://microservice-provider-user/" + id, User.class);
  }

  /**
   * hystrix fallback方法
   * @param id id
   * @return 默认的用户
   */
  public User fallback(Long id) {
    RibbonHystrixService.LOGGER.info("异常发生，进入fallback方法，接收的参数：id = {}", id);
    User user = new User();
    user.setId(-1L);
    user.setUsername("default username");
    user.setAge(0);
    return user;
  }
}
```

controller：RibbonHystrixController.java

```
@RestController
public class RibbonHystrixController {
  @Autowired
  private RibbonHystrixService ribbonHystrixService;

  @GetMapping("/ribbon/{id}")
  public User findById(@PathVariable Long id) {
    return this.ribbonHystrixService.findById(id);
  }
}
```

application.yml

```
server:
  port: 8011
spring:
  application:
    name: microservice-consumer-movie-ribbon-with-hystrix
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    hostname: ribbon          # 此处，preferIpAddress不设置或者设为false，不能设为true，否则影响turbine的测试。turbine存在的问题：eureka.instance.hostname一致时只能检测到一个节点，会造成turbine数据不完整
```

验证：

1. 启动注册中心：microservice-discovery-eureka
2. 启动服务提供方：microservice-provider-user
3. 启动服务消费方：microservice-consumer-movie-ribbon-with-hystrix
4. 访问：http://localhost:8011/ribbon/1，获得结果：`{"id":1,"username":"Tom","age":12}`

5. 关闭服务提供方：microservice-provider-user，访问 http://localhost:8011/ribbon/1，获得结果：`{"id":-1,"username":"default username","age":0}`，另外日志打印：`c.i.c.s.u.service.RibbonHystrixService : 异常发生，进入fallback方法，接收的参数：id = 1`。

注意：

> 1. 本示例代码在microservice-consumer-movie-ribbon基础上修改而来
> 2. 如对本示例涉及的知识点有疑难，请查看上一章《服务消费者》

代码地址（任选其一）：

> 1. http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon-with-hystrix
> 2. https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-ribbon-with-hystrix

Feign使用Hystrix

代码示例

在Feign中使用Hystrix是非常简单的事情，因为Feign已经集成了Hystrix。我们使用microservice-consumer-movie-feign-with-hystrix项目的代码做一点修改，将其中的UserClient.java修改为如下即可：

```
/**
 * 使用@FeignClient注解的fallback属性，指定fallback类
 * @author eacdy
 */
@FeignClient(name = "microservice-provider-user", fallback = Hys
trixClientFallback.class)
public interface UserFeignHystrixClient {
  @RequestMapping("/{id}")
  public User findByIdFeign(@RequestParam("id") Long id);

  /**
   * 这边采取了和Spring Cloud官方文档相同的做法，将fallback类作为内部类
放入Feign的接口中，当然也可以单独写一个fallback类。
   * @author eacdy
   */
  @Component
  static class HystrixClientFallback implements UserFeignHystrix
Client {
    private static final Logger LOGGER = LoggerFactory.getLogger
(HystrixClientFallback.class);

    /**
     * hystrix fallback方法
     * @param id id
     * @return 默认的用户
     */
    @Override
    public User findByIdFeign(Long id) {
      HystrixClientFallback.LOGGER.info("异常发生，进入fallback方法
，接收的参数：id = {}", id);
      User user = new User();
      user.setId(-1L);
      user.setUsername("default username");
      user.setAge(0);
      return user;
    }
  }
}
```

这样就完成了，是不是很简单呢？测试过程类似通用方式。

注意：

> 1. 本示例代码在microservice-consumer-movie-feign基础上修改而来
> 2. 如对本示例涉及的知识点有疑难，请查看上一章《服务消费者》

代码地址（任选其一）

> 1. http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix
> 2. https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix

参考文档：

> 1. https://msdn.microsoft.com/en-us/library/dn589784.aspx
> 2. http://martinfowler.com/bliki/CircuitBreaker.html
> 3. http://particular.net/blog/protect-your-software-with-the-circuit-breaker-design-pattern
> 4. https://github.com/Netflix/Hystrix/wiki/How-it-Works#flow-chart
> 5. https://segmentfault.com/a/1190000005988895

2.4.2. Hystrix Dashboard

Hystrix监控

除了隔离依赖服务的调用以外，Hystrix还提供了近实时的监控，Hystrix会实时、累加地记录所有关于HystrixCommand的执行信息，包括每秒执行多少请求多少成功，多少失败等。Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。

上文提到的 `microservice-consumer-movie-ribbon-with-hystrix` 项目已经具备对Hystrix监控的能力，下面我们进入测试。

测试步骤

1. 启动：microservice-discovery-eureka
2. 启动：microservice-provider-user
3. 启动：microservice-consumer-movie-ribbon-with-hystrix
4. 访问：http://localhost:8011/ribbon/1，注意：该步骤不能省略，因为如果应用的所有接口都未被调用，将只会看到一个ping
5. 访问：http://localhost:8011/hystrix.stream，可以看到类似如下输出：

```
data: {"type":"HystrixCommand","name":"findById","group":"RibbonHystrixService","currentTime":1472658867784,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":0,"rollingCountBadRequests":0....}
```

并且会不断刷新以获取实时的监控数据。但是纯文字的输出可读性实在是太差，运维人员很难一眼看出系统当前的运行状态。那么是不是有可视化的工具呢？

Hystrix Dashboard

Hystrix Dashboard可以可视化查看实时监控数据。我们可以下载hystrix-dashboard的war包部署到诸如Tomcat之类的容器中，本文不做赘述。另外Spring Cloud也提供了Hystrix Dashboard的整合，下面我们看看Spring Cloud是怎么玩转Hystrix Dashboard的。

新建一个maven项目，在pom.xml中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-hystrix-dashboard</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-hystrix-dashboard</artifa
ctId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

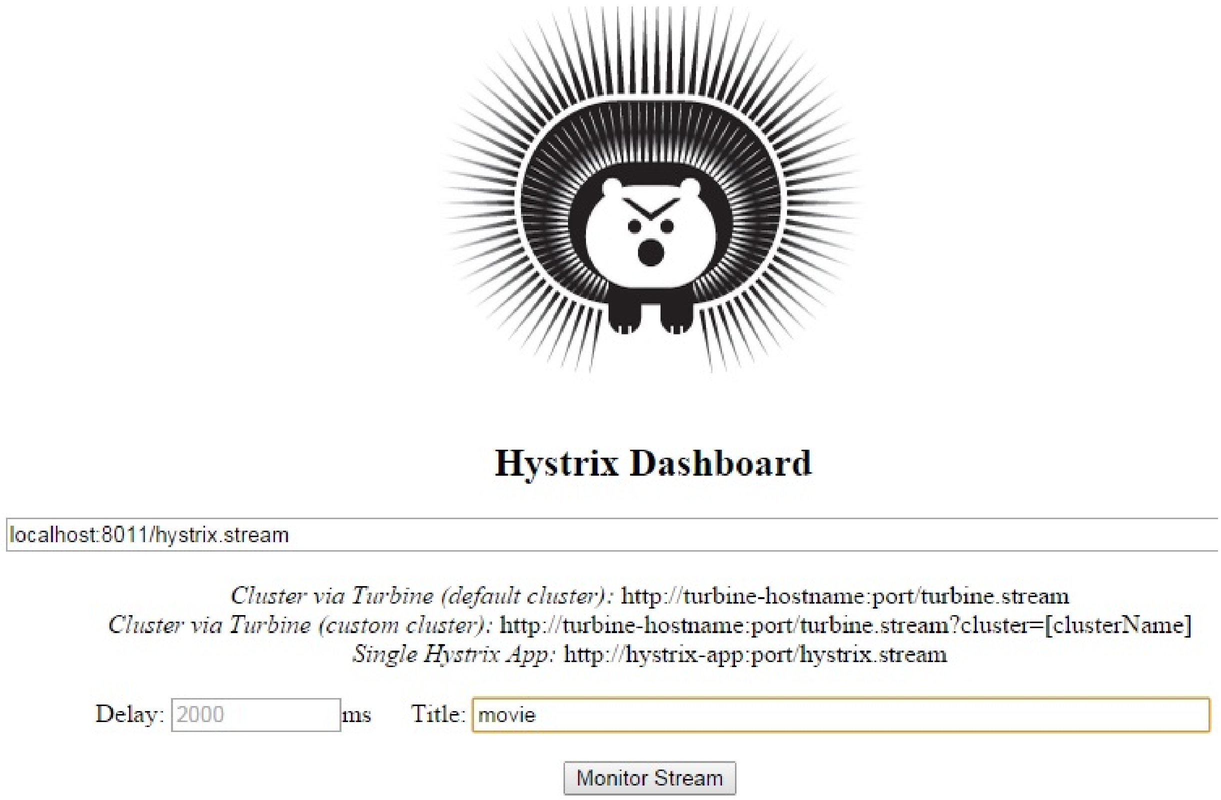
编写启动类：HystrixDashboardApplication.java

```
/**
 * 测试步骤:
 * 1. 访问http://localhost:8030/hystrix.stream 可以查看Dashboard
 * 2. 在上面的输入框填入: http://想监控的服务:端口/hystrix.stream进行测
试
 * 注意：首先要先调用一下想监控的服务的API，否则将会显示一个空的图表.
 * @author eacdy
 */
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
  public static void main(String[] args) {
    new SpringApplicationBuilder(HystrixDashboardApplication.cla
ss).web(true).run(args);
  }
}
```

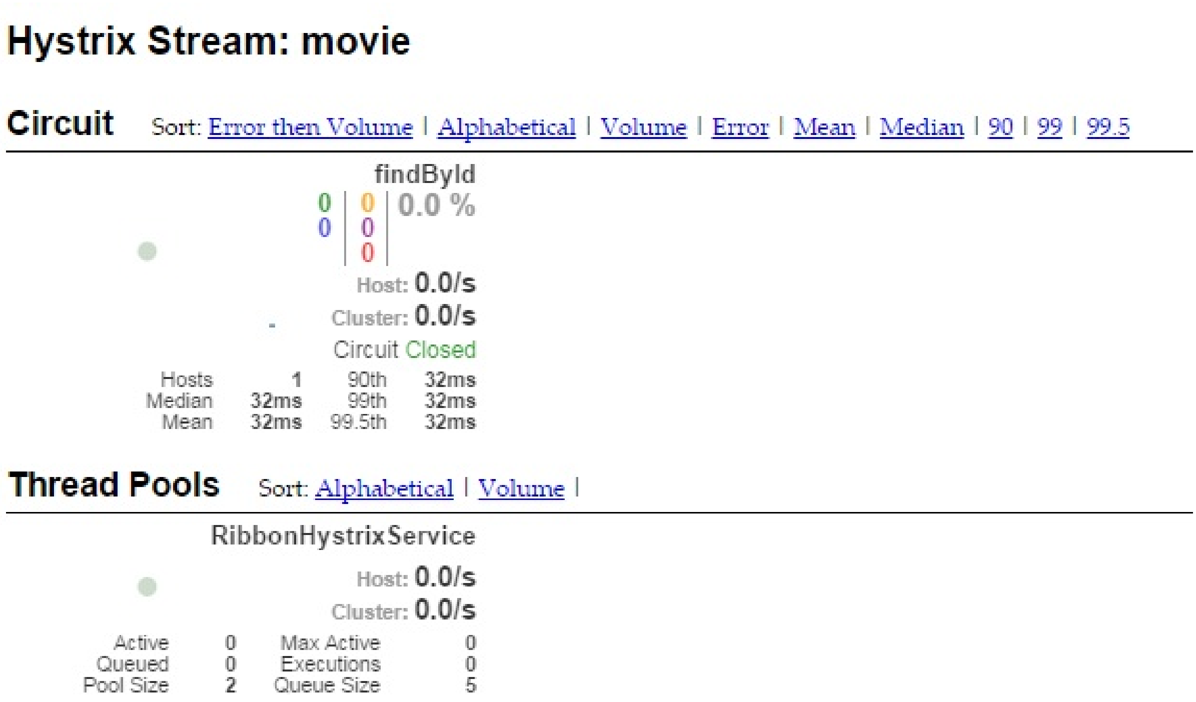
配置文件：application.yml

```
spring:
  application:
    name: hystrix-dashboard
server:
  port: 8030
```

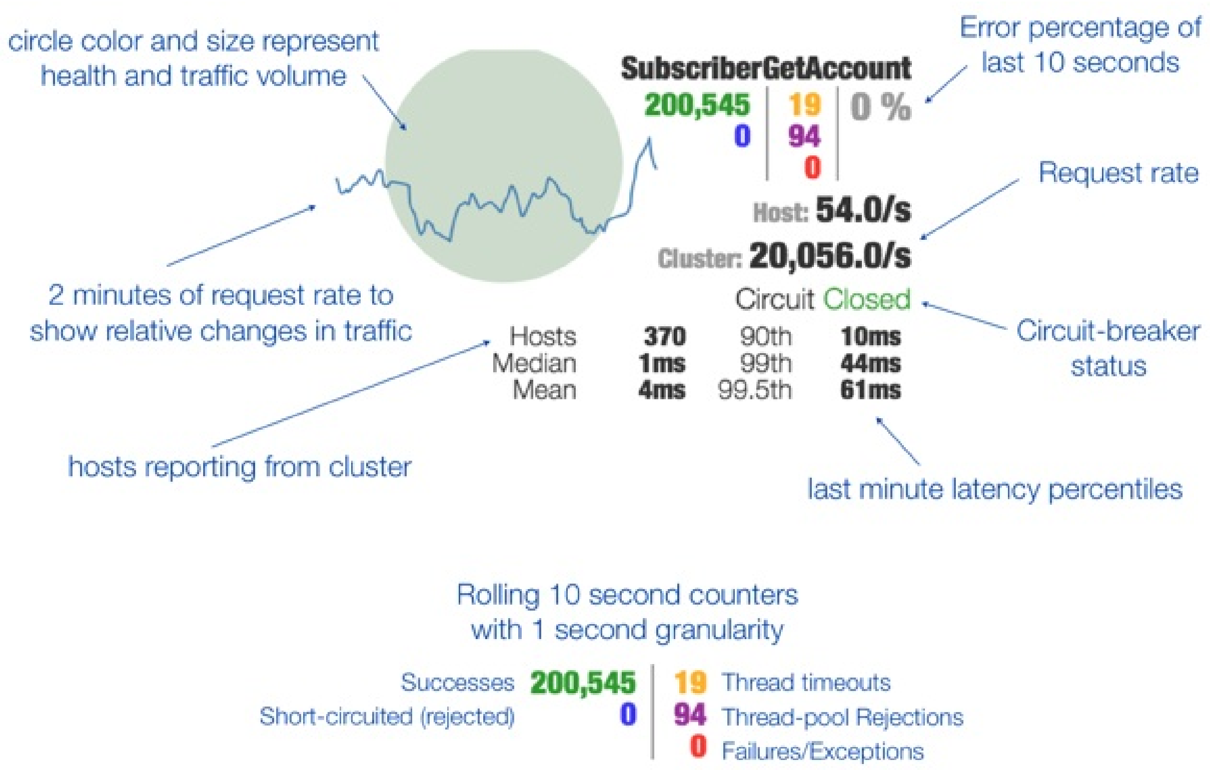
启动后，访问http://localhost:8030/hystrix.stream将会看到如下界面：



此时，我们在输入框中输入http://localhost:8011/hystrix.stream，并随意设置一个Title后，点击Monitor Stream按钮，会出现如下界面：



此时我们会看到findById这个API的各种指标。Hystrix Dashboard Wiki上详细说明了图上每个指标的含义，如下图：



此时，我们可以尝试将microservice-provider-user停止，然后重复访问多次http://localhost:8011/ribbon/1（20次以上），会发现断路器状态会变为开启。

代码地址（任选其一）：

> http://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-hystrix-dashboard
>
> https://github.com/eacdy/spring-cloud-study/tree/master/microservice-hystrix-dashboard

TIPS

1. Hystrix的监控数据默认是保存在每个实例的内存中的，Spring Boot提供了多种方式，可以导入到Redis、TSDB以供日后分析使用。
2. 我们启动前文的 `microservice-consumer-movie-feign-with-hystrix` 项目后发现其访问localhost:8021/hystrix.stream ，是404，查看pom.xml依赖树，发现其没有依赖hystrix-metrics-event-stream项目。故而添加依赖：

```
<!-- 整合hystrix，其实feign中自带了hystrix，引入该依赖主要是为了使用其中的hystrix-metrics-event-stream，用于dashboard -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

并在启动类上添加 `@EnableCircuitBreaker` 注解即可。详见项目 `microservice-consumer-movie-feign-with-hystrix-stream` ，因为这不是本文的讨论重点，故而只做扩展阅读。

`microservice-consumer-movie-feign-with-hystrix-stream` 代码地址（二选一）：

> https://github.com/eacdy/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix-stream
>
> https://git.oschina.net/itmuch/spring-cloud-study/tree/master/microservice-consumer-movie-feign-with-hystrix-stream

2.4.3. Turbine

Turbine

在复杂的分布式系统中，相同服务的节点经常需要部署上百甚至上千个，很多时候，运维人员希望能够把相同服务的节点状态以一个整体集群的形式展现出来，这样可以更好的把握整个系统的状态。 为此，Netflix提供了一个开源项目（Turbine）来提供把多个hystrix.stream的内容聚合为一个数据源供Dashboard展示。

和Hystrix Dashboard一样，Turbine也可以下载war包部署到Web容器，本文不做赘述。下面讨论Spring Cloud是怎么使用Turbine的。

代码示例

新建Maven项目，并在pom.xml中添加如下内容：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-hystrix-turbine</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-turbine</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-netflix-turbine</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类：TurbineApplication.java

```
/**
 * 通过@EnableTurbine接口，激活对Turbine的支持。
 * @author eacdy
 */
@SpringBootApplication
@EnableTurbine
public class TurbineApplication {
  public static void main(String[] args) {
    new SpringApplicationBuilder(TurbineApplication.class).web(true).run(args);
  }
}
```

配置文件：application.yml

```
spring:
  application.name: microservice-hystrix-turbine
server:
  port: 8031
security.basic.enabled: false
turbine:
  aggregator:
    clusterConfig: default   # 指定聚合哪些集群，多个使用","分割，默认为default。可使用http://.../turbine.stream?cluster={clusterConfig之一}访问
  appConfig: microservice-consumer-movie-feign-with-hystrix-stream,microservice-consumer-movie-ribbon-with-hystrix  ### 配置Eureka中的serviceId列表，表明监控哪些服务
  clusterNameExpression: new String("default")
  # 1. clusterNameExpression指定集群名称，默认表达式appName；此时：turbine.aggregator.clusterConfig需要配置想要监控的应用名称
  # 2. 当clusterNameExpression: default时，turbine.aggregator.clusterConfig可以不写，因为默认就是default
  # 3. 当clusterNameExpression: metadata['cluster']时，假设想要监控的应用配置了eureka.instance.metadata-map.cluster: ABC，则需要配置，同时turbine.aggregator.clusterConfig: ABC
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/


### 参考：http://blog.csdn.net/liaokailin/article/details/51344281

### 参考：http://blog.csdn.net/zhuchuangang/article/details/51289593
```

这样一个Turbine微服务就编写完成了。

Turbine测试

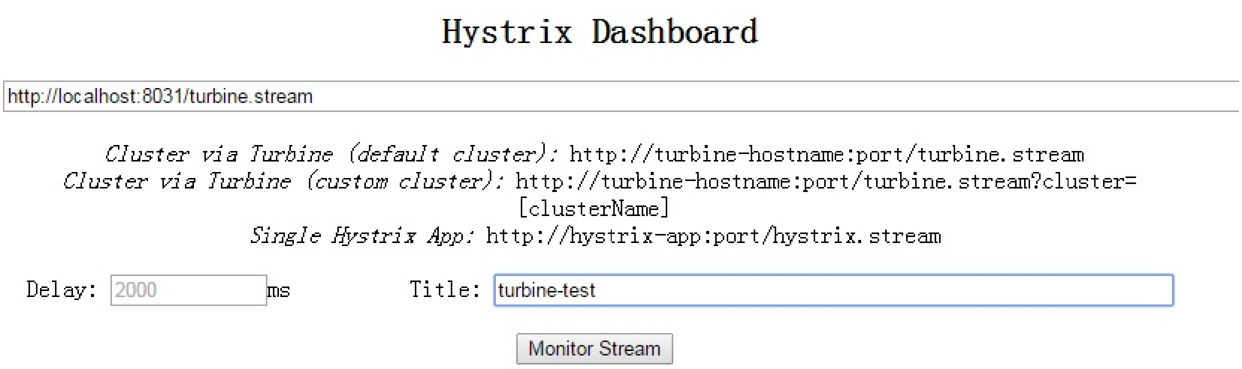
1. 启动项目：microservice-discovery-eureka

2. 启动项目：microservice-provider-user
3. 启动项目：microservice-consumer-movie-ribbon-with-hystrix
4. 启动项目：microservice-consumer-movie-feign-with-hystrix-stream
5. 启动项目：microservice-hystrix-dashboard
6. 启动项目：microservice-hystrix-turbine（即本例）
7. 访问：http://localhost:8011/ribbon/1，调用ribbon接口
8. 访问：http://localhost:8022/feign/1，调用feign接口
9. 访问：http://localhost:8031/turbine.stream，可查看到和Hystrix监控类似的内容：

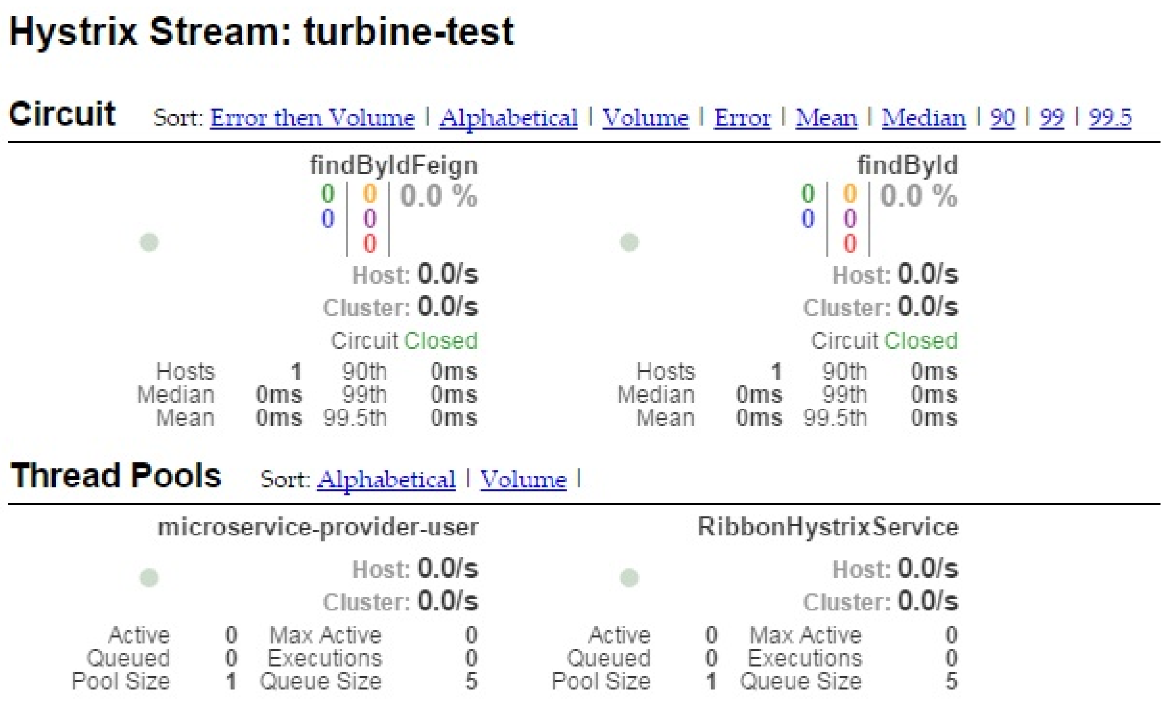
```
data: {"rollingCountFallbackSuccess":0,"rollingCountFallbackFail
ure":0,"propertyValue_circuitBreakerRequestVolumeThreshold":20,"
p
```

并且会不断刷新以获取实时的监控数据。同样的，我们可以将这些文本内容放入到Dashboard中展示。

访问Hystrix Dashboard：http://localhost:8030/hystrix.stream，并将http://localhost:8031/turbine.stream输入到其上的输入框，并随意指定一个Title，如下图：



将会查看到如下的图表，有图可见，我们把两个项目的API都监控了：



TIPS

1. 项目 `microservice-consumer-movie-ribbon-with-hystrix` 和 `microservice-consumer-movie-feign-with-hystrix-stream` 需要配置不同的主机名，并将preferIpAddress设为false或者不设置，否则将会造成在单个主机上测试，Turbine只显示一个图表的情况。项目的代码已经修改好并注释了，这边友情提示一下。

2. 配置项： `turbine.clusterNameExpression` 与 `turbine.aggregator.clusterConfig` 的关系：

turbine.clusterNameExpression取值	turbine.aggregator.clusterConfig 取值
默认（appName）	配置想要聚合的项目，此时使用turbine.stream?cluster=项目名称大写访问监控数据
new String("default") 或者"'default'"	不配置，或者配置default，因为默认就是default
metadata['cluster']；同时待监控的项目配置了类似：eureka.instance.metadata-map.cluster: ABC	也设成ABC，需要和待监控的项目配置的eureka.instance.metadata-map.cluster一致。

具体可以关注 `org.springframework.cloud.netflix.turbine.CommonsInstanceDiscovery` 和 `org.springframework.cloud.netflix.turbine.EurekaInstanceDiscovery` 两个类。特别关注一下 `org.springframework.cloud.netflix.turbine.EurekaInstanceDiscovery.marshall(InstanceInfo)` 方法。

参考文档

> http://blog.csdn.net/liaokailin/article/details/51344281
>
> http://stackoverflow.com/questions/31468227/whats-for-the-spring-cloud-turbine-clusternameexpression-config-mean

2.5 配置中心

Spring Cloud Config提供了一种在分布式系统中外部化配置服务器和客户端的支持。配置服务器有一个中心位置，管理所有环境下的应用的外部属性。客户端和服务器映射到相同Spring Eventment 和 PropertySrouce抽象的概念，所以非常适合Spring应用，但也可以在任何语言开发的任何应用中使用。在一个应用从开发、测试到生产的过程中，你可以分别地管理开发、测试、生产环境的配置，并且在迁移的时候获取相应的配置来运行。

Config Server 存储后端默认使用git存储配置信息，因此可以很容易支持标记配置环境的版本，同时可以使用一个使用广泛的工具管理配置内容。当然添加其他方式的存储实现也是很容易的。

参考：

> http://cloud.spring.io/spring-cloud-static/Brixton.SR5/#_spring_cloud_config

核心代码：

> `org.springframework.cloud.config.server.environment.NativeEnvironmentRepository` ，留意其中的 `findOne` 方法。

代码示例

准备工作

- 为了更贴近生产，我们首先配置Host

```
127.0.0.1 config-server
```

- 准备几个配置文件，命名规范为 `项目名称-环境名称.properties` ，本文在git仓库：https://github.com/eacdy/spring-cloud-study/中，新建目录config-repo，创建以下几个文件：

```
microservice-config-client-dev.properties
microservice-config-client.properties
```

其中在： `microservice-config-client-dev.properties` 文件中添加如下内容：

```
profile=dev
```

服务器端代码示例

创建一个Maven项目，在pom.xml文件中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-config-server</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类：

```
/**
 * 通过@EnableConfigServer注解激活配置服务.
 * 说明：
 * 在application.yml中有个git.uri的配置，目前配置的是https://github.c
om/eacdy/spring-cloud-study/
 * 获取git上的资源信息遵循如下规则：
 * /{application}/{profile}[/{label}]
 * /{application}-{profile}.yml
 * /{label}/{application}-{profile}.yml
 * /{application}-{profile}.properties
 * /{label}/{application}-{profile}.properties
 *
 * 例如本例：可使用以下路径来访问microservice-config-client-dev.prope
rties：
 * http://localhost:8040/microservice-config-client-dev.properti
es
 * http://localhost:8040/microservice-config-client/dev
 * ...
 * @author eacdy
 */
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServerApplication.class, args);
  }
}
```

配置文件：application.yml

```
server:
  port: 8040
spring:
  application:
    name: microservice-config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/eacdy/spring-cloud-study/
# 配置git仓库的地址
          search-paths: config-repo
# git仓库地址下的相对地址，可以配置多个，用,分割。
          username:
# git仓库的账号
          password:
# git仓库的密码
```

这样，一个Config Server就完成了。

测试工作

获取**git**上的资源信息遵循如下规则
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties

例如本例：可使用以下路径来访问 `microservice-config-client-dev.properties` ：http://localhost:8040/microservice-config-client-dev.properties http://localhost:8040/microservice-config-client/dev

按照上文，我们成功搭建了Config Server，并测试能够正常获取到git仓库中的配置信息。那么对于一个微服务应用，如何才能获取配置信息呢？

配置服务客户端示例

新建一个Maven项目，在pom.xml中添加如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-config-client</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

编写启动类：

```java
@SpringBootApplication
public class ConfigClientApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigClientApplication.class, args);
  }
}
```

编写测试Controller

```java
/**
 * 这边的@RefreshScope注解不能少，否则即使调用/refresh，配置也不会刷新
 * @author eacdy
 */
@RestController
@RefreshScope
public class ConfigClientController {
  @Value("${profile}")
  private String profile;

  @GetMapping("/hello")
  public String hello() {
    return this.profile;
  }
}
```

配置文件：application.yml

```yaml
server:
  port: 8041
```

配置文件bootstrap.yml（为什么要使用**bootstrap.yml**而不直接放在**application.yml**中的原因见注意点）

```
spring:
  application:
    name: microservice-config-client    # 对应microservice-config-server所获取的配置文件的{application}
  cloud:
    config:
      uri: http://config-server:8040/
      profile: dev                      # 指定profile，对应microservice-config-server所获取的配置文件中的{profile}
      label: master                     # 指定git仓库的分支，对应microservice-config-server所获取的配置文件的{label}
```

启动，并访问：

http://localhost:8041/hello ，我们会发现此时可以显示git仓库中配置文件的内容：

```
dev
```

配置内容的热加载

如果不重启应用，能够做到配置的刷新吗？答案显然是可以的。

我们将microservice-config-client-dev.properties中值改为

```
profile=abcd
```

并提交到git仓库。

然后使用命令（本文使用的是curl，Linux和Windows都有curl工具，当然也可以借助其他工具，例如Postman等）：

```
curl  -X POST http://localhost:8041/refresh
```

然后再次访问http://localhost:8041/hello ，将会看到： `abcd` ，说明配置已经刷新。

配置服务与注册中心联合使用

在生产环境中，我们可能会将Config Server 与 Eureka等注册中心联合使用（注意：目前Spring Cloud只支持与Eureka及Consul联合使用，不支持与Zookeeper联合使用），下面讲解如何将Config Server与 Eureka 联合使用。

准备工作

1. 启动服务 `microservice-discovery-eureka` ；
2. 和上文一样，准备好几个配置文件，命名规范为 `项目名称-环境名称.properties` ，本文使用的名称是 `microservice-config-client-eureka-dev.properties` 。

代码示例

服务器端代码示例：

首先新建一个Maven项目，在 `pom.xml` 中添加如下内容：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-config-server-eureka</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类： `ConfigServerEurekaApplication.java`

```
@SpringBootApplication
@EnableConfigServer
@EnableDiscoveryClient
public class ConfigServerEurekaApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServerEurekaApplication.class, a
rgs);
  }
}
```

配置文件：`application.yml`

```
server:
  port: 8050
spring:
  application:
    name: microservice-config-server-eureka
  cloud:
    config:
      server:
        git:
          uri: https://github.com/eacdy/spring-cloud-study/
          search-paths: config-repo
          username:
          password:
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
```

客户端示例

创建一个Maven项目，在 `pom.xml` 中添加如下内容：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-config-client-eureka</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类： `ConfigClientEurekaApplication.java`

```
@SpringBootApplication
@EnableDiscoveryClient
public class ConfigClientEurekaApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConfigClientEurekaApplication.class, a
rgs);
  }
}
```

编写测试Controller

```
/**
 * 这边的@RefreshScope注解不能少，否则即使调用/refresh，配置也不会刷新
 * @author eacdy
 */
@RestController
@RefreshScope
public class ConfigClientController {
  @Value("${profile}")
  private String profile;

  @GetMapping("/hello")
  public String hello() {
    return this.profile;
  }
}
```

配置文件：application.yml

```
server:
  port: 8051
```

配置文件：bootstrap.yml

```
spring:
  application:
    name: microservice-config-client-eureka
  cloud:
    config:
      profile: dev
      label: master
      discovery:
        enabled: true                                  # 默认false，设为true表示使用注册中心中的configserver配置而不自己配置configserver的uri
        serviceId: microservice-config-server-eureka  # 指定config server在服务发现中的serviceId，默认为：configserver
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/

# 参考文档：https://github.com/spring-cloud/spring-cloud-config/blob/master/docs/src/main/asciidoc/spring-cloud-config.adoc#discovery-first-bootstrap
```

从示例代码我们发现，想要将Config Server 与 注册中心联合使用，只需要在客户端侧配置 `spring.cloud.config.discovery.enabled=true` 和 `spring.cloud.config.discovery.serviceId` 两个配置项即可。Eureka的配置前文有讲到过，如有疑问，详见服务发现的相关章节。

注意：当服务发现是 `Eureka` 及 `Consul` 时，Config Server支持与之联合使用；如果是 `Zookeeper` 做服务发现，目前不支持与之联合使用。

注意点：

- client需要添加以下依赖，否则访问/refresh将会得到404：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- client的controller需要添加@RefreshScope注解，否则配置无法刷新。
- 本文的 `bootstrap.yml` 文件中的内容不能放到 `application.yml` 中，否则config部分无法被加载，因为config部分的配置先于 `application.yml` 被加载，而 `bootstrap.yml` 中的配置会先于 `application.yml` 加载，
- Config Server也可以支持本地存储或svn而不使用git，相对较为简单，故而本文不作赘述，有兴趣的可以自行阅读Spring Cloud的文档。

参考文档：

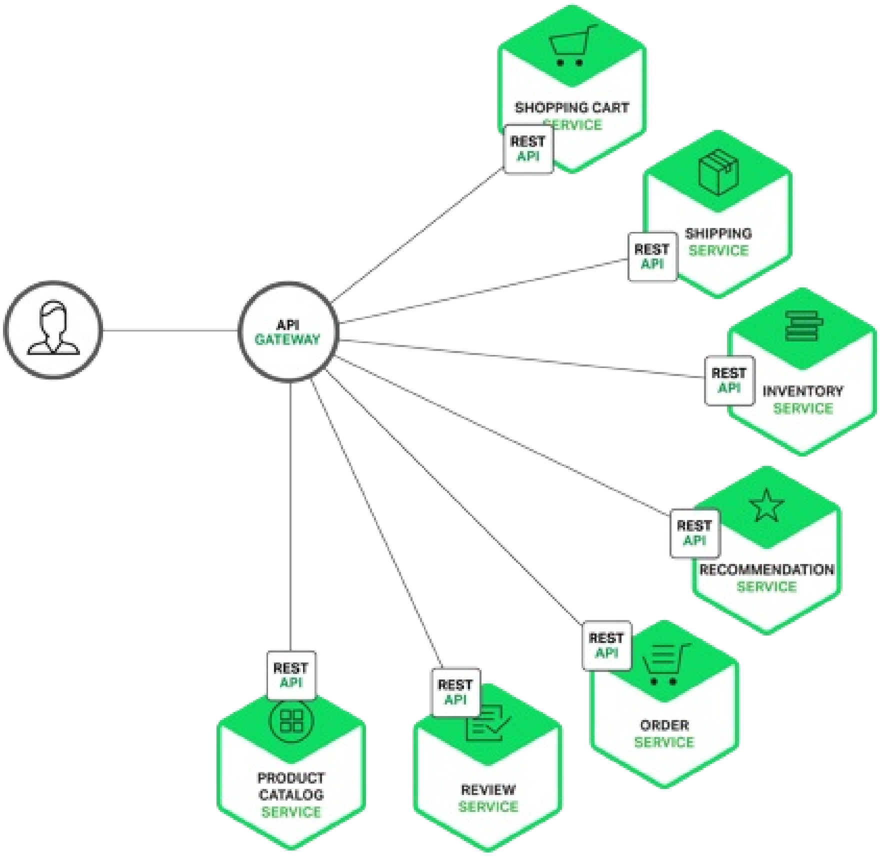
> Config Server与注册中心联合使用：https://github.com/spring-cloud/spring-cloud-config/blob/master/docs/src/main/asciidoc/spring-cloud-config.adoc#discovery-first-bootstrap
>
> Config Server的高可用： https://github.com/spring-cloud/spring-cloud-config/issues/87

2.6 API Gateway

API Gateway是微服务架构中不可或缺的部分。API Gateway的定义以及存在的意义，Chris已经为大家描述过了，本文不再赘述，以下是链接：

> 中文版：http://dockone.io/article/482
>
> 英文版：https://www.nginx.com/blog/building-microservices-using-an-api-gateway/

使用API Gateway后，客户端和微服务之间的网络图变成下图：



通过API Gateway，可以统一向外部系统提供REST API。Spring Cloud中使用Zuul作为API Gateway。Zuul提供了动态路由、监控、回退、安全等功能。

下面我们进入Zuul的学习：

准备工作

- 为了更贴近生产，我们首先配置Host

```
127.0.0.1 gateway
```

- 启动服务：microservice-discovery-eureka
- 启动服务：microservice-provider-user

Zuul代码示例

创建Maven项目，在pom.xml中添加如下内容：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://m
aven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-api-gateway</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-zuul</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类：

```
/**
 * 使用@EnableZuulProxy注解激活zuul。
 * 跟进该注解可以看到该注解整合了@EnableCircuitBreaker、@EnableDiscov
eryClient，是个组合注解，目的是简化配置。
 * @author eacdy
 */
@SpringBootApplication
@EnableZuulProxy
public class ZuulApiGatewayApplication {
  public static void main(String[] args) {
    SpringApplication.run(ZuulApiGatewayApplication.class, args)
;
  }
}
```

配置文件：application.yml

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
eureka:
  instance:
    hostname: gateway
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
```

这样，一个简单的API Gateway就完成了。

测试

启动microservice-api-gateway项目。还记得我们之前访问通过 http://localhost:8000/1去访问microservice-provider-user服务中id=1的用户信息吗？

我们现在访问http://localhost:8050/microservice-provider-user/1试试。会惊人地看到：

```
{"id":1,"username":"Tom","age":12}
```

这不正是microservice-provider-user服务中id=1的用户信息吗？

所以我们可以总结出规律：访问

```
http://GATEWAY:GATEWAY_PORT/想要访问的Eureka服务id的小写/**
```

，将会访问到

```
http://想要访问的Eureka服务id的小写:该服务端口/**
```

自定义路径

上文我们已经完成了通过API Gateway去访问微服务的目的，是通过

```
http://GATEWAY:GATEWAY_PORT/想要访问的Eureka服务id的小写/**
```

的形式访问的，那么如果我们想自定义在API Gateway中的路径呢？譬如想使用

```
http://localhost:8050/user/1
```

就能够将请求路由到http://localhost:8000/1呢？

只需要做一点小小的配置即可：

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
eureka:
  instance:
    hostname: gateway
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
# 下面整个树都非必须，如果不配置，将默认使用 http://GATEWAY:GATEWAY_PORT/想要访问的Eureka服务id的小写/** 路由到：http://想要访问的Eureka服务id的小写:该服务端口/**
zuul:
  routes:
    user:                                                # 可以随便写，在zuul上面唯一即可；当这里的值 = service-id时，service-id可以不写。
      path: /user/**                                     # 想要映射到的路径
      service-id: microservice-provider-user             # Eureka中的serviceId
```

如何忽略某些服务

准备工作

1. 启动服务：microservice-discovery-eureka
2. 启动服务：microservice-provider-user
3. 启动服务：microservice-consumer-movie-ribbon

如果我们现在只想将microservice-consumer-movie-ribbon服务暴露给外部，microservice-provider-user不想暴露，那么应该怎么办呢？

依然只是一点小小的配置即可：

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
eureka:
  instance:
    hostname: gateway
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
zuul:
  ignored-services: microservice-provider-user          # 需要忽视的服务(配置后将不会被路由)
  routes:
    movie:                                                # 可以随便写，在zuul上面唯一即可；当这里的值 = service-id时，service-id可以不写。
      path: /movie/**                                     # 想要映射到的路径
      service-id: microservice-consumer-movie-ribbon-with-hystrix     # Eureka中的serviceId
```

这样microservice-provider-user服务就不会被路由，microservice-consumer-movie-ribbon服务则会被路由。

测试结果：

URL	结果	
http://localhost:8050/microservice-provider-user/1	404	说 mic pro 未
http://localhost:8050/movie/ribbon/1	{"id":1,"username":"Tom","age":12}	说 mic con mo 被

使用Zuul不使用Eureka

Zuul并不依赖Eureka，可以脱离Eureka运行，此时需要配置

```
spring:
  application:
    name: microservice-api-gateway
server:
  port: 8050
zuul:
  routes:
    movie:                                                # 可以随便写
      path: /user/**
      url: http://localhost:8000/                         # path路由到的地址，也就是访问http://localhost:8050/user/**会路由到http://localhost:8000/**
```

我们可尝试访问http://localhost:8050/user/1 ，会发现被路由到了 http://localhost:8000/1 。不过在大多数情况下，笔者并不建议这么做，因为得手动大量地配置URL，不是很方便。

其他使用

Zuul还支持更多的特性、更多的配置项甚至是定制开发，具体还请读者自行发掘。

同类软件

Zuul只是API Gateway的一种实现，可作为API Gateway的软件有很多，譬如Nginx Plus、Kong等等，本文不做赘述了。

参考文档

https://www.nginx.com/blog/building-microservices-using-an-api-gateway/
http://microservices.io/patterns/apigateway.html

3 使用Docker构建微服务

准备工作

安装软件	版本	功能	必要程度
Docker	1.12.1	Docker	是
CentOS7.0或其他系统	7.0	Docker的宿主机，本章的讲解都是在CentOS 7.0下进行的。Docker现已支持Windows系统，但考虑到绝大多数Docker容器还是跑在Linux环境下的，故而只讲解Linux环境下的使用。Windows下的安装使用大致类似，请读者自行研究。	是
Maven	3.3.9		是
JDK	8u65		是

注意：

> 本章的讲解都是在CentOS7下进行的，建议新手使用CentOS 7.x进入学习。

CentOS下JDK 1.8的安装

(1) 到Oracle官网下载好 `jdk-8u65-linux-x64.rpm` 备用

(2) 卸载系统自带java

```
java -version           # 如果有结果出来，则说明自带了java
rpm -qa|grep java       # 查询出已经安装的java
yum -y remove [上面查出来的东西，多个用空格分隔]
```

(3) 安装JDK

```
cd /usr
mkdir /usr/java
rpm -ivh jdk-8u65-linux-x64.rpm
```

(4) 配置环境变量：

```
vim /etc/profile
```

找到： `export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL` 这一行，并在其下面一行添加如下内容：

```
# 设置java环境变量
export JAVA_HOME=/usr/java/jdk1.8.0_65 # 根据情况修改
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.ja
r
```

(5) 使环境变量生效：

```
source /etc/profile
```

(6) 测试：

```
java -version
javac
```

CentOS下Maven的安装

Maven的安装比较简单，只需要下载后解压，配置环境变量即可。

(1) 下载并解压：

```
cd /opt
wget http://apache.fayea.com/maven/maven-3/3.3.9/binaries/apache
-maven-3.3.9-bin.tar.gz
tar -zxvf apache-maven-3.3.9-bin.tar.gz
```

(2) 配置环境变量：

```
vim /etc/profile
```

找到 `export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL` ，并在其下面一行添加如下内容：

```
# 设置Maven环境变量
export MAVEN_HOME=/opt/apache-maven-3.3.9/
export PATH=$MAVEN_HOME/bin:$PATH
```

(3) 使环境变量生效：

```
source /etc/profile
```

(4) 测试

```
mvn -version
```

(5) 修改Maven配置

本地仓库路径配置：

```
<!-- 本地仓库路径配置 -->
<localRepository>/path/to/local/repo</localRepository>
```

国内Maven镜像配置：

```
<mirror>
  <id>aliyun</id>
  <mirrorOf>central</mirrorOf>
  <name>Aliyun Central mirror</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

CentOS下Git的安装

```
yum install git
```

3.1 Docker介绍

Docker概览

Docker是一个用于开发、交付和运行应用的开放平台，Docker被设计用于更快地交付应用。Docker可以将应用程序和基础设施层隔离，并且可以将基础设施当作程序一样进行管理。使用Docker，可以更快地打包代码、测试以及部署，并且可以减少从编写到部署运行代码的周期。

Docker将内核容器特性（LXC）、工作流和工具集成，以帮助管理和部署应用。

什么是Docker

核心是，Docker了一种在安全隔离的容器中运行近乎所有应用的方式，这种隔离性和安全性允许你在同一个主机上同时运行多个容器，而容器的这种轻量级特性，无需消耗运行hpervisor所需的额外负载，意味着你可以节省更多的硬件资源。

基于容器虚拟化的工具或平台可提供：

- 将应用（包括支撑组件）放入Docker容器中
- 分发和交付这些容器给团队，便于后续的开发和测试
- 将容器部署到生产环境中，生产环境可以是本地的数据中心，也可以在云端。

自从上世纪 90 年代硬件虚拟化被主流的技术广泛普及之后，对数据中心而言，发生的最大的变革莫过于容器和容器管理工具，例如：Docker。在过去的一年内，Docker 技术已经逐渐走向成熟，并且推动了大型初创公司例如 Twitter 和 Airbnb 的发展，甚至在银行、连锁超市、甚至 NASA 的数据中心都赢得了一席之地。当我几年前第一次直到 Docker 的时候，我还对 Docker 的未来持怀疑的态度，我认为他们是把以前的 Linux 容器的概念拿出来包装了一番推向市场。但是使用 Docker 成功进行了几个项目 例如 Spantree 之后，我改变了我的看法：Docker 帮助我们节省了大量的时间和经历，并且已经成为我们技术团队中不可或缺的工具。 GitHub 上面每天都会催生出各式各样的工具、形态各异的语言和千奇百怪的概念。如果你和我一样，没有时间去把他们全部都测试一遍，甚至没有时间去亲自测试 Docker，那么你可以看一下我的这篇文章：我将会用我们在 Docker 中总结的经验来告诉你什么是 Docker、为什么 Docker 会这么火。

Docker 是容器管理工具 Docker 是一个轻量级、便携式、与外界隔离的容器，也是一个可以在容器中很方便地构建、传输、运行应用的引擎。和传统的虚拟化技术不同的是，Docker 引擎并不虚拟出一台虚拟机，而是直接使用宿主机的内核和硬件，直接在宿主机上运行容器内应用。也正是得益于此，Docker 容器内运行的应用和宿主机上运行的应用性能差距几乎可以忽略不计。 但是 Docker 本身并不是一个容器系统，而是一个基于原有的容器化工具 LXC 用来创建虚拟环境的工具。类似 LXC 的工具已经在生产环境中使用多年，Docker 则基于此提供了更加友好的镜像管理工具和部署工具。

Docker 不是虚拟化引擎 Docker 第一次发布的时候，很多人都拿 Docker 和虚拟机 VMware、KVM 和 VirtualBox 比较。尽管从功能上看，Docker 和虚拟化技术致力于解决的问题都差不多，但是 Docker 却是采取了另一种非常不同的方式。虚拟机是虚拟出一套硬件，虚拟机的系统进行的磁盘操作，其实都是在对虚拟出来的磁盘进行操作。当运行 CPU 密集型的任务时，是虚拟机把虚拟系统里的 CPU 指令“翻译”成宿主机的CPU指令并进行执行。两个磁盘层，两个处理器调度器，两个操作系统消耗的内存，所有虚拟出的这些都会带来相当多的性能损失，一台虚拟机所消耗的硬件资源和对应的硬件相当，一台主机上跑太多的虚拟机之后就会过载。而 Docker 就没有这种顾虑。Docker 运行应用采取的是“容器”的解决方案：使用 namespace 和 CGroup 进行资源限制，和宿主机共享内核，不虚拟磁盘，所有的容器磁盘操作其实都是对 /var/lib/docker/ 的操作。简言之，Docker 其实只是在宿主机中运行了一个受到限制的应用程序。 从上面不难看出，容器和虚拟机的概念并不相同，容器也并不能取代虚拟机。在容器力所不能及的地方，虚拟机可以大显身手。例如：宿主机是 Linux，只能通过虚拟机运行 Windows，Docker 便无法做到。再例如，宿主机是 Windows，Windows 并不能直接运行 Docker，Windows上的 Docker 其实是运行在 VirtualBox 虚拟机里的。

Docker 使用层级的文件系统 前面提到过，Docker 和现有容器技术 LXC 等相比，优势之一就是 Docker 提供了镜像管理。对于 Docker 而言，镜像是一个静态的、只读的容器文件系统的快照。然而不仅如此，Docker 中所有的磁盘操作都是对特定的Copy-On-Write文件系统进行的。下面通过一个例子解释一下这个问题。 例如我们要建立一个容器运行 JAVA Web 应用，那么我们应该使用一个已经安装了 JAVA 的镜像。在 Dockerfile（一个用于生成镜像的指令文件）中，应该指明“基于 JAVA 镜像”，这样 Docker 就会去 Docker Hub Registry 上下载提前构建好的 JAVA 镜像。然后再 Dockerfile 中指明下载并解压 Apache Tomcat 软件到 /opt/tomcat 文件夹中。这条命令并不会对原有的 JAVA 镜像产生任何影响，而仅仅是在原有镜像上面添加了一个改动层。当一个容器启动时，容器内的所有改动层都会启动，容器会从第一层中运行 /usr/bin/java 命令，并且调用另外一层中的 /opt/tomcat/bin 命

令。实际上，Dockerfile 中每一条指令都会产生一个新的改动层，即便只有一个文件被改动。如果用过 Git 就能更清楚地认识这一点，每条指令就像是每次 commit，都会留下记录。但是对于 Docker 来说，这种文件系统提供了更大的灵活性，也可以更方便地管理应用程序。 我们Spantree的团队有一个自己维护的含有 Tomcat 的镜像。发布新版本也非常简单：使用 Dockerfile 将新版本拷贝进镜像从而创建一个新镜像，然后给新镜像贴上版本的标签。不同版本的镜像的不同之处仅仅是一个 90 MB 大小的 WAR 文件，他们所基于的主镜像都是相同的。如果使用虚拟机去维护这些不同的版本的话，还要消耗掉很多不同的磁盘去存储相同的系统，而使用 Docker 就只需要很小的磁盘空间。即便我们同时运行这个镜像的很多实例，我们也只需要一个基础的 JAVA / TOMCAT 镜像。

Docker 可以节约时间 很多年前我在为一个连锁餐厅开发软件时，仅仅是为了描述如何搭建环境都需要写一个 12 页的 Word 文档。例如本地 Oracle 数据库，特定版本的 JAVA，以及其他七七八八的系统工具和共享库、软件包。整个搭建过程浪费掉了我们团队每个人几乎一天的时间，如果用金钱衡量的话，花掉了我们上万美金的时间成本。虽然客户已经对这种事情习以为常，甚至认为这是引入新成员、让成员适应环境、让自己的员工适应我们的软件所必须的成本，但是相比较起来，我们宁愿把更多的时间花在为客户构建可以增进业务的功能上面。 如果当时有 Docker，那么构建环境就会像使用自动化搭建工具 Puppet / Chef / Salt / Ansible 一样简单，我们也可以把整个搭建时间周期从一天缩短为几分钟。但是和这些工具不同的地方在于，Docker 可以不仅仅可以搭建整个环境，还可以将整个环境保存成磁盘文件，然后复制到别的地方。需要从源码编译 Node.js 吗？Docker 做得到。Docker 不仅仅可以构建一个 Node.js 环境，还可以将整个环境做成镜像，然后保存到任何地方。当然，由于 Docker 是一个容器，所以不用担心容器内执行的东西会对宿主机产生任何的影响。 现在新加入我们团队的人只需要运行 docker-compose up 命令，便可以喝杯咖啡，然后开始工作了。

Docker 可以节省开销 当然，时间就是金钱。除了时间外，Docker 还可以节省在基础设施硬件上的开销。高德纳和麦肯锡的研究表明，数据中心的利用率在 6% - 12% 左右。不仅如此，如果采用虚拟机的话，你还需要被动地监控和设置每台虚拟机的 CPU 硬盘和内存的使用率，因为采用了静态分区(static partitioning)所以资源并不能完全被利用。。而容器可以解决这个问题：容器可以在实例之间进行内存和磁盘共享。你可以在同一台主机上运行多个服务、可以不用去限制容器所消耗的资源、可以去限制资源、可以在不需要的时候停止容器，也不用担心启动已经停止的程序时会带来过多的资源消耗。凌晨三点的时候只有很少的人会去访问你的网站，同时你需要比较多的资源执行夜间的批处理任务，那么可以很简单的便实现资源的交换。 虚拟机所消耗的内存、硬盘、CPU 都是固定的，一般动态调整都需要重启

虚拟机。而用 Docker 的话，你可以进行资源限制，得益于 CGroup，可以很方便动态调整资源限制，让然也可以不进行资源限制。Docker 容器内的应用对宿主机而言只是两个隔离的应用程序，并不是两个虚拟机，所以宿主机也可以自行去分配资源。

注：

> 部分翻译自：https://docs.docker.com/engine/understanding-docker/
>
> 部分参考：http://zhidao.baidu.com/link?url=4FOwNhnpVC3FP0hOxaC4vrl3fFG27lWRpDEaZ3KJBVL0E29C5O-ty4zqze1On52Uk4kcNrnPd3VEKpKvRs4pNEV-lgo78lmP1_FXffMerdG

参考文档

Docker介绍：http://www.lupaworld.com/article-243555-1.html

Docker介绍：http://www.docker.org.cn/book/docker/what-is-docker-16.html

Docker官方文档：https://docs.docker.com/engine/understanding-docker/

Docker中文文档：http://git.oschina.net/widuu/chinese_docker

Docker介绍：https://segmentfault.com/a/1190000002609286

3.2 Docker的安装

Docker的安装是比较简单的，笔者原本不想过多提及；但是看到有不少读者对Docker的安装提出了疑问，故此进行一个安装的总结。

对于Linux用户可以借助其发行版的Linux包管理工具安装，对于Windows和MAC用户相对麻烦一些，笔者下面以Windows7系统为例，讲述安装过程。笔者强烈建议大家使用**Linux**系统进入本章的学习，第一是比较符合目前Docker的市场趋势，第二Docker本身就是基于Linux的LXC技术。

CentOS 7.0下Docker的安装

1. 查看内核版本(Docker需要64位版本，同时内核版本在3.10以上，如果版本低于3.10，需要升级内核)：

    ```
    uname -r
    ```

2. 更新yum包：

    ```
    yum update
    ```

3. 添加yum仓库：

    ```
    sudo tee /etc/yum.repos.d/docker.repo <<-'EOF'
    [dockerrepo]
    name=Docker Repository
    baseurl=https://yum.dockerproject.org/repo/main/centos/7/
    enabled=1
    gpgcheck=1
    gpgkey=https://yum.dockerproject.org/gpg
    EOF
    ```

4. 安装Docker

```
yum install docker-engine
```

5. 启动Docker

```
service docker start
```

6. 使用Docker国内镜像（为Docker镜像下载提速，非必须）

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | s
h -s http://fe8a7d6e.m.daocloud.io
```

参考：

> 官方文档：https://docs.docker.com/engine/installation/linux/centos/

CentOS 6.5下Docker的安装

Docker容器最早受到RHEL完善的支持是从最近的CentOS 7.0开始的，官方说明是只能运行于64位架构平台，内核版本为2.6.32-431及以上（即 >= CentOS 6.5，运行docker时实际提示3.10.0及以上）。需要注意的是CentOS 6.5与7.0的安装是有一点点不同的，CentOS 6.x上Docker的安装包叫docker-io，并且来源于Fedora epel库，这个仓库维护了大量的没有包含在发行版中的软件，所以先要安装EPEL，而CentOS 7.x的Docker直接包含在官方镜像源的Extras仓库（CentOS-Base.repo下的[extras]节enable=1启用）。

下面就CentOS 6.5讲解Docker的安装过程，以下是软件版本：

Linux版本	Docker版本
CentOS 6.5 X64（只能X64）	1.7.1

升级内核

查看内核版本：

```
uname -r
```

结果：`2.6.32-431.el6.x86_64`，不满足上文的需求，故此需要升级内核。

升级步骤：

1. 导入公钥数字证书

```
rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org
```

2. 安装ELRepo

```
rpm -ivh http://www.elrepo.org/elrepo-release-6-5.el6.elrepo
.noarch.rpm
```

3. 安装kernel长期版本

```
yum --enablerepo=elrepo-kernel install kernel-lt -y     # lt
表示long-term的意思，长期维护版本，也可以将kernel-lt改为kernel-ml，
安装主线版本
```

4. 编辑grub.conf文件，修改Grub引导顺序，确认刚安装好的内核在哪个位置，然后设置default值（从0开始），一般新安装的内核在第一个位置，所以设置default=0。

```
vim /etc/grub.conf

# 以下是/etc/grub.conf的内容
default=0        # 修改该值即可
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title CentOS (3.10.103-1.el6.elrepo.x86_64)
```

5. 重启并查看内核版本，将会发现内核已经更新。

安装Docker

1. 禁用selinux，因为selinux和LXC有冲突，故而需要禁用

```
vim /etc/selinux/config的内容

# 以下是/etc/selinux/config的内容
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforc
ing.
#     disabled - No SELinux policy is loaded.
SELINUX=disabled  # 将SELINUX设为disabled，注意修改后最好重启下机
器。
```

2. 安装 Fedora EPEL

```
yum -y install http://dl.fedoraproject.org/pub/epel/6/x86_64
/epel-release-6-8.noarch.rpm
```

3. 安装Docker

```
yum install -y docker-io
```

4. 以守护模式运行Docker

```
docker -d
```

5. 如果不报错，那就是启动成功了，如果报以下异常：

```
docker: relocation error: docker: symbol dm_task_get_info_wi
th_deferred_remove, version Base not defined in file libdevm
apper.so.1.02 with link time reference
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
```

执行以下内容：

```
yum upgrade device-mapper-libs
```

6. 将Docker开机启动

```
chkconfig docker on
```

7. 重启机器

其他平台的安装

请参考：https://docs.docker.com/engine/installation/

参考文档

> Windows：https://docs.docker.com/engine/installation/windows/
>
> MAC： https://docs.docker.com/engine/installation/mac/
>
> CentOS：https://docs.docker.com/engine/installation/linux/centos/

3.3 Docker的常用命令

准备工作

1. 对于Window用户，请点击Kitematic左下方的DOCKER CLI按钮，在弹出的命令窗体内输入命令，不要在CMD中测试Docker命令。
2. 下载镜像，以kitematic/hello-world-nginx为例：

```
docker pull kitematic/hello-world-nginx
```

常用命令测试一览表

命令	解释
docker images	列表本地所有镜像
docker search 关键词	在Docker Hub中搜索镜像
docker pull 镜像名称	下载Docker镜像
docker rmi 镜像id	删除Docker镜像。加参数-f表示强制删除。
docker run 镜像名称	启动Docker镜像
docker ps	列表所有运行中的Docker容器。该命令参数比较多，-a：列表所有容器；-f：过滤；-q 只列表容器的id。
docker version	查看Docker版本信息
docker info	查看Docker系统信息，例如：CPU、内存、容器个数等等
docker kill 容器id	杀死id对应容器
docker start / stop / restart 容器id	启动、停止、重启指定容器
docker build -t 标签名称 目录	构建Docker镜像，-t 表示指定一个标签
docker tag	为镜像打标签

更多命令，请输入 `--help` 参数查询；如果想看docker命令可输入 `docker --help` ；如果想查询 `docker run` 命令的用法，可输入 `docker run --help` 。

docker run

docker run 应该是我们最常用的命令了，这边讲解一下，便于大家入门。

参数	解释
-d	后台运行
-P	随机端口映射
-p	指定端口映射 格式： ip:hostPort:containerPort ip::containerPort hostPort:containerPort containerPort

测试：

1. 启动测试镜像 `docker pull kitematic/hello-world-nginx`

   ```
   docker run -d -p 91:80 kitematic/hello-world-nginx
   ```

   这边解释下docker run的两个参数：

   ```
   -d                              # 后台运行
   -p 宿主机端口:容器端口          # 开放容器端口到宿主机端口
   ```

1. 访问：http://localhost:91 测试，这里的localhost指的是宿主机的主机名



说明

由于Docker的入门不是本文章的主要话题，同时入门也是非常简单的，所以不做太多赘述了，我们尽量把话题聚焦在微服务上。如果有疑问的童鞋可以给我提Issue，也可以等待我的“Docker手把手”系列文章。

TIPS

> 1. 目前网上很多教程还是boot2docker（项目地址：https://github.com/boot2docker/boot2docker），该项目已废弃了，笔者不建议使用。
> 2. 如果安装不成功，请按照提示解决一下，譬如开启VT技术等等。Docker的提示是做得非常好的，如果还是安装不上，请给我提Issue，可以远程解决。
> 3. 对于Winodws用户，建议点击Kitematic左下角的DOCKER CLI，在弹出来的控制台输入Docker命令，而不要直接在CMD中输入命令。

参考文档：

> http://my.oschina.net/denglz/blog/487332
> https://segmentfault.com/a/1190000000733628
> http://www.oschina.net/translate/nstalling-dockerio-on-centos-64-64-bit
> https://segmentfault.com/a/1190000000735011
> http://www.server110.com/docker/201411/11122.html
> http://dockone.io/article/152 http://www.open-open.com/lib/view/open1422492851548.html
>
> Docker常用命令：http://www.infoq.com/cn/articles/docker-command-line-quest/

3.4 Dockerfile常用指令

指令的一般格式为 `指令名称 参数` 。

FROM

支持三种格式：

- `FROM <image>`
- `FROM <image>:<tag>`
- `FROM <image>@<digest>`

FROM指令必须指定且需要在Dockerfile其他指令的前面，指定的基础image可以是官方远程仓库中的，也可以位于本地仓库。后续的指令都依赖于该指令指定的image。当在同一个Dockerfile中建立多个镜像时，可以使用多个FROM指令。

MAINTAINER

格式为：

- `MAINTAINER <name>`

用于指定维护者的信息。

RUN

支持两种格式：

- `RUN <command>`
- 或 `RUN ["executable", "param1", "param2"]` 。

`RUN <command>` 在shell终端中运行命令，在Linux中默认是 `/bin/sh -c` 在Windows中是 `cmd /s /c` `RUN ["executable", "param1", "param2"]` 使用exec执行。指定其他终端可以通过该方式操作，例如： `RUN ["/bin/bash", "-c", "echo hello"]` ，该方式必须使用["]而不能使用[']，因为该方式会被转换成一个JSON 数组。

CMD

支持三种格式：

`CMD ["executable","param1","param2"]` (推荐使用)

`CMD ["param1","param2"]` (为ENTRYPOINT指令提供预设参数)

`CMD command param1 param2` (在shell中执行)

CMD指令的主要目的是为执行容器提供默认值。每个Dockerfile只有一个CMD命令，如果指定了多个CMD命令，那么只有一条会被执行，如果启动容器的时候指定了运行的命令，则会覆盖掉CMD指定的命令。

LABEL

格式为：

- `LABEL <key>=<value> <key>=<value> <key>=<value> ...`

为镜像添加元数据。使用 "和 \ 转换命令行，示例：

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

EXPOSE

格式为：

- `EXPOSE <port> [<port>...]`

为Docker容器设置对外的端口号。在启动时，可以使用-p选项或者-P选项。

示例：

```
# 映射一个端口示例
EXPOSE port1
# 相应的运行容器使用的命令
docker run -p port1 image
# 也可以使用-P选项启动
docker run -P image

# 映射多个端口示例
EXPOSE port1 port2 port3
# 相应的运行容器使用的命令
docker run -p port1 -p port2 -p port3 image
# 还可以指定需要映射到宿主机器上的某个端口号
docker run -p host_port1:port1 -p host_port2:port2 -p host_port3
:port3 image
```

ENV

格式为：

- `ENV <key> <value>`
- `ENV <key>=<value> ...`

指定环境变量，会被后续RUN指令使用，并在容器启动后，可以通过 `docker inspect` 查看这个环境变量，也可以通过 `docker run --env <key>=<value>` 来修改环境变量

示例：

```
ENV JAVA_HOME /path/to/java # 设置环境变量JAVA_HOME
```

ADD

格式为：

- `ADD <src>... <dest>`
- `ADD ["<src>",... "<dest>"]`

从src目录复制文件到容器的dest。其中src可以是Dockerfile所在目录的相对路径，也可以是一个URL，还可以是一个压缩包

注意：

1. src必须在构建的上下文内，不能使用例如： `ADD ../somethine /something` ，因为 `docker build` 命令首先会将上下文路径和其子目录发送到docker daemon
2. 如果src是一个URL，同时dest不以斜杠结尾，dest将会被视为文件，src对应内容文件将会被下载到dest
3. 如果src是一个URL，同时dest以斜杠结尾，dest将被视为目录，src对应内容将会被下载到dest目录
4. 如果src是一个目录，那么整个目录其下的内容将会被拷贝，包括文件系统元数据
5. 如果文件是可识别的压缩包格式，则docker会自动解压

COPY

格式为：

- `COPY <src>... <dest>`
- `COPY ["<src>",... "<dest>"]` （shell中执行）

复制本地端的src到容器的dest。和ADD指令类似，COPY不支持URL和压缩包。

ENTRYPOINT

格式为：

- `ENTRYPOINT ["executable", "param1", "param2"]`
- `ENTRYPOINT command param1 param2`

指定Docker容器启动时执行的命令，可以多次设置，但是只有最后一个有效。

VOLUME

格式为：

- `VOLUME ["/data"]`

使容器中的一个目录具有持久化存储数据的功能，该目录可以被容器本身使用，也可以共享给其他容器。当容器中的应用有持久化数据的需求时可以在Dockerfile中使用该指令。

USER

格式为：

- `USER 用户名`

设置启动容器的用户，默认是root用户。

WORKDIR

格式为：

- `WORKDIR /path/to/workdir`

切换目录指令，类似于cd命令，对RUN、CMD、ENTRYPOINT生效。

ARG

格式为：

- `ARG <name>[=<default value>]`

ARG指令定义一个变量。

ONBUILD

格式为：

- `ONBUILD [INSTRUCTION]`

指定当建立的镜像作为其他镜像的基础时，所执行的命令。

其他

STOPSINGAL HEALTHCHECK SHELL 由于并不是很常用，所以不做讲解了。有兴趣的可以前往https://docs.docker.com/engine/reference/builder/ 扩展阅读。

参考文档：

Dockerfile文档：https://docs.docker.com/engine/reference/builder/#dockerfile-reference

Dockerfile最佳实践：https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#build-cache

Docker书籍：http://udn.yyuap.com/doc/docker_practice/advanced_network/port_mapping.html

Docker书籍：https://philipzheng.gitbooks.io/docker_practice/content/dockerfile/instructions.html

Dockerfile讲解：http://blog.csdn.net/qinyushuang/article/details/43342553

Dockerfile讲解：http://blog.csdn.net/wsscy2004/article/details/25878223

Dockerfile网络：http://my.oschina.net/ghm7753/blog/522809

COPY 和 ADD 的区别：http://blog.163.com/digoal@126/blog/static/163877040201410341236664/

CMD与ENTRYPOINT的区别：http://cloud.51cto.com/art/201411/457338.htm

3.5 Docker私有仓库的搭建与使用

和Maven一样，Docker不仅提供了一个中央仓库，同时也允许我们搭建私有仓库。如果读者对Maven有所了解，将会很容易理解私有仓库的优势：

- 节省带宽，镜像无需从中央仓库下载，只需从私有仓库中下载即可
- 对于私有仓库中已有的镜像，提升了下载速度
- 便于内部镜像的统一管理

下面我们来讲解一下如何搭建、使用私有仓库

准备工作

准备两台安装有Docker的CentOS7的机器，主机规划如下（仅供参考）：

主机	IP	角色
node0	192.168.11.143	Docker开发机
node1	192.168.11.144	Docker私有仓库

安装、使用私有仓库

网上有很多 `docker-registry` 的教程，但是 `docker-registry` 已经过时，并且已经2年不维护了。详见https://github.com/docker/docker-registry ，故而本文不做探讨，对 `docker-registry` 有兴趣的童鞋可以查阅本节的参考文档。

本节讲解registry V2，registry V2需要Docker版本高于1.6.0。registry V2要求使用https访问，那么我们先做一些准备，为了方便，这边模拟以域名 `reg.itmuch.com` 进行讲解。

使用域名搭建https的私有仓库

- 首先修改两台机器的hosts，配置 `192.168.11.144` 到 `reg.itmuch.com` 的映射

```
echo '192.168.11.144 reg.itmuch.com'>> /etc/hosts
```

- 既然使用https，那么我们需要生成证书，本文讲解的是使用openssl自签名证书，当然也可以使用诸如 `Let’s Encrypt` 等工具生成证书，首先在node1机器上生成key：

```
mkdir -p ~/certs
cd ~/certs
openssl genrsa -out reg.itmuch.com.key 2048
```

再生成密钥文件：

```
openssl req -newkey rsa:4096 -nodes -sha256 -keyout reg.itmuch.c
om.key -x509 -days 365 -out reg.itmuch.com.crt
```

会有一些信息需要填写：

```
Country Name (2 letter code) [XX]:CN
        # 你的国家名称
State or Province Name (full name) []:JS
        # 省份
Locality Name (eg, city) [Default City]:NJ
          # 所在城市
Organization Name (eg, company) [Default Company Ltd]:ITMUCH
        # 组织名称
Organizational Unit Name (eg, section) []:ITMUCH
        # 组织单元名称
Common Name (eg, your name or your server's hostname) []:reg.itm
uch.com # 域名
Email Address []:eacdy0000@126.com
          # 邮箱
```

这样自签名证书就制作完成了。

- 由于是自签名证书，默认是不受Docker信任的，故而需要将证书添加到Docker的根证书中，Docker在CentOS 7中，证书存放路径

是 `/etc/docker/certs.d/域名` ：

`node1` 端：

```
mkdir -p /etc/docker/certs.d/reg.itmuch.com
cp ~/certs/reg.itmuch.com.crt /etc/docker/certs.d/reg.itmuch.com
/
```

`node0` 端：将生成的证书下载到根证书路径

```
mkdir -p /etc/docker/certs.d/reg.itmuch.com
scp root@192.168.11.144:/root/certs/reg.itmuch.com.crt /etc/dock
er/certs.d/reg.itmuch.com/
```

- 重新启动 `node0` 和 `node1` 的Docker

```
service docker restart
```

- 在 `node1` 上启动私有仓库

首先切换到家目录中，这一步不能少，原因是下面的-v 挂载了证书，如果不切换，将会引用不到证书文件。

```
cd ~
```

启动Docker私有仓库（注意：如果直接粘贴运行，请删除掉注释）：

```
docker run -d -p 443:5000 --restart=always --name registry \
  -v `pwd`/certs:/certs \                                  # 将“当
前目录/certs”挂载到容器的“/certs”
  -v /opt/docker-image:/opt/docker-image \

  -e STORAGE_PATH=/opt/docker-image \
     # 指定容器内存储镜像的路径
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/reg.itmuch.com.crt \
      # 指定证书文件
  -e REGISTRY_HTTP_TLS_KEY=/certs/reg.itmuch.com.key \
      # 指定key文件
  registry:2
```

其中，之所以挂载/opt/docker-image目录，是为了防止私有仓库容器被删除，私有仓库中的镜像也会丢失。

- 在 `node0` 上测试，将镜像push到私服

```
docker pull kitematic/hello-world-nginx
docker tag kitematic/hello-world-nginx reg.itmuch.com/kitematic/
hello-world-nginx # 为本地镜像打标签
docker push reg.itmuch.com/kitematic/hello-world-nginx
  # 将镜像push到私服
```

会发现如下内容：

```
The push refers to a repository [reg.itmuch.com/kitematic/hello-
world-nginx]
5f70bf18a086: Pushed
b51acdd3ef48: Pushed
3f47ff454588: Pushed
....
latest: digest: sha256:d3e1883b703c39556f2f09da14cc3b820f69a4343
6655c882c0c0ded0dda6a4b size: 3226
```

说明已经push成功。

- 从私服中下载镜像：

```
docker pull reg.itmuch.com/kitematic/hello-world-nginx
```

配置登录认证

在很多场景下，我们需要用户登录后才能访问私有仓库，那么我们可以如下操作：

建立在上文生成证书，同时重启过Docker服务的前提下，我们讲解一下如何配置：

- 为防止端口冲突，我们首先删除或停止之前启动好的私有仓库：

```
docker kill registry
```

- 在node1机器上安装 `httpd-tools` ：

```
yum install httpd-tools
```

- 在node机器上创建密码文件，并添加一个用户 `testuser` ，密码是 `testpassword` ：

```
cd ~
mkdir auth
htpasswd -Bbn testuser testpassword > auth/htpasswd
```

- 在node1机器上切换到 `~` 目录，并启动私有仓库（注意：如果直接粘贴运行，请删除掉注释）：

```
docker run -d -p 443:5000 --restart=always --name registry2 \
  -v /opt/docker-image:/var/lib/registry \
      # 挂载容器内存储镜像路径到宿主机
  -v `pwd`/certs:/certs \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/reg.itmuch.com.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/reg.itmuch.com.key \
  -v `pwd`/auth:/auth \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
  registry:2
```

- 测试：

```
docker push reg.itmuch.com/kitematic/hello-world-nginx
```

提示：

```
461f75075df2: Image push failed
no basic auth credentials
```

说明需要认证。

我们登陆一下，执行：

```
docker login reg.imuch.com
```

再次执行

```
docker push reg.itmuch.com/kitematic/hello-world-nginx
```

就可以正常push镜像到私有仓库了。

注意：如果想要从私有仓库上下载镜像，同样需要登录。

参考文档：

官方文档：https://docs.docker.com/registry/deploying/#/running-a-domain-registry

Docker Registry V2 htpasswd认证方式搭建：http://www.tuicool.com/articles/vMZZveM

Docker Registry V2搭建：http://www.tuicool.com/articles/6jEJZj

Docker Registry V2搭建：http://tomhat.iteye.com/blog/2304098

Docker Registry V1搭建：http://blog.csdn.net/wsscy2004/article/details/26279569

非认证的Docker Registry V1搭建：http://blog.csdn.net/wangtaoking1/article/details/44180901

带认证的Docker Registry V1搭建：http://snoopyxdy.blog.163.com/blog/static/601174402015823741997/

Docker专题汇总：http://www.zimug.com/360.html

Docker疑难解答：https://segmentfault.com/q/1010000000938076

3.6 使用Dockerfile构建Docker镜像

下面我们以microservice-discovery-eureka项目为例，我们首先执行

```
mvn clean package # 使用Maven打包项目
```

将项目构建成jar包： `microservice-discovery-eureka-0.0.1-SNAPSHOT.jar` ，那么如果我们想要启动项目则只需要在 `microservice-discovery-eureka-0.0.1-SNAPSHOT.jar` 所在的目录（即项目的target目录）执行：

```
java -jar microservice-discovery-eureka-0.0.1-SNAPSHOT.jar
```

那么如果我们现在想要将项目在Docker容器中运行，需要怎么做呢？

使用Dockerfile构建Docker镜像

- 在 `microservice-discovery-eureka-0.0.1-SNAPSHOT.jar` 所在目录（默认即：项目构建后的target目录，当然也可以将jar文件拷贝到其他任意路径），创建文件，命名为Dockerfile

```
# 基于哪个镜像
FROM java:8

# 将本地文件夹挂载到当前容器
VOLUME /tmp

# 拷贝文件到容器，也可以直接写成ADD microservice-discovery-eureka-0.0
.1-SNAPSHOT.jar /app.jar
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'

# 开放8761端口
EXPOSE 8761

# 配置容器启动后执行的命令
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-j
ar","/app.jar"]
```

- 构建docker镜像，执行：

```
docker build -t eacdy/test1 .        # 格式：docker build -t 标签
名称 Dockerfile的相对位置
```

构建成功： `Successfully built a7cc6f4de088` 。

- 启动镜像

```
docker run -p 8761:8761 eacdy/test1
```

- 访问 `http://Docker宿主机IP:8761` ，我们会发现Eureka能够正常被访问。

参考文档

> 基于Dockerfile搭建JAVA Tomcat运行环境：http://www.blogjava.net/yongboy/archive/2013/12/16/407643.html
>
> Docker实现增量发布之前期准备：http://blog.csdn.net/zssureqh/article/details/52009043
>
> Dockerfile详解：http://blog.csdn.net/wsscy2004/article/details/25878223
>
> Dockerfile RUN/CMD/ENTRYPOINT命令详解：http://blog.163.com/digoal@126/blog/static/163877040201410411715832/
>
> 如何使用Dockerfile构建镜像：http://blog.csdn.net/qinyushuang/article/details/43342553

3.7 使用Maven插件构建Docker镜像

工具

工欲善其事，必先利其器。笔者经过调研，有以下几款Docker的Maven插件进入笔者视野：

插件名称	官方地址
docker-maven-plugin	https://github.com/spotify/docker-maven-plugin
docker-maven-plugin	https://github.com/fabric8io/docker-maven-plugin
docker-maven-plugin	https://github.com/bibryam/docker-maven-plugin

笔者从Stars、文档易用性以及更新频率三个纬度考虑，选用了第一款。

使用插件构建Docker镜像

简单使用

我们以之前的项目：microservice-discovery-eureka为例：

- 在pom.xml中添加下面这段

```
    <build>
        <plugins>
            <!-- docker的maven插件，官网：https://github.com/spotify/docker-maven-plugin -->
            <plugin>
                <groupId>com.spotify</groupId>
                <artifactId>docker-maven-plugin</artifactId>
                <version>0.4.12</version>
                <configuration>
                    <!-- 注意imageName一定要是符合正则[a-z0-9-_.]的，否则构建不会成功 -->
                    <!-- 详见：https://github.com/spotify/docker-maven-plugin    Invalid repository name ... only [a-z0-9-_.] are allowed-->
                    <imageName>microservice-discovery-eureka</imageName>
                    <baseImage>java</baseImage>
                    <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>
                    <resources>
                        <resource>
                            <targetPath>/</targetPath>
                            <directory>${project.build.directory}</directory>
                            <include>${project.build.finalName}.jar</include>
                        </resource>
                    </resources>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

- 执行命令：

```
mvn clean package docker:build
```

- 我们会发现控制台有类似如下内容：

```
[INFO] Building image microservice-discovery-eureka
Step 1 : FROM java
Pulling from library/java
Digest: sha256:581a4afcbbedd8fdf194d597cb5106c1f91463024fb3a49a2
d9f025165eb675f
Status: Downloaded newer image for java:latest
 ---> ea40c858f006
Step 2 : ADD /microservice-discovery-eureka-0.0.1-SNAPSHOT.jar /
/
 ---> d1c174083bca
Removing intermediate container 91913d847c20
Step 3 : ENTRYPOINT java -jar /microservice-discovery-eureka-0.0
.1-SNAPSHOT.jar
 ---> Running in 0f2aeccdfd46
 ---> d57b027ca65a
Removing intermediate container 0f2aeccdfd46
Successfully built d57b027ca65a
[INFO] Built microservice-discovery-eureka
[INFO] ----------------------------------------------------------
---------------
[INFO] BUILD SUCCESS
[INFO] ----------------------------------------------------------
---------------
[INFO] Total time: 01:38 min
[INFO] Finished at: 2016-09-18T01:05:05-07:00
[INFO] Final Memory: 40M/198M
```

恭喜，构建成功了。

- 我们执行 `docker images` 会发现该镜像已经被构建成功：

```
REPOSITORY                        TAG                 IMAGE ID
        CREATED             SIZE
microservice-discovery-eureka   latest              d57b027ca65a
        About a minute ago   681.5 MB
```

- 启动镜像

```
docker run -p 8761:8761 microservice-discovery-eureka
```

我们会发现该Docker镜像会很快地启动。

- 访问测试

访问http://Docker宿主机IP:8761 ，能够正常看到Eureka界面。

使用Dockerfile进行构建

上文讲述的方式是最简单的方式，很多时候，我们还是要借助Dockerfile进行构建的，首先我们在/microservice-discovery-eureka/src/main/docker目录下，建立文件Dockerfile

```
FROM java:8
VOLUME /tmp
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
EXPOSE 9000
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

修改pom.xml

```
<build>
        <plugins>
            <!-- docker的maven插件，官网：https://github.com/spoti
fy/docker-maven-plugin -->
            <plugin>
                <groupId>com.spotify</groupId>
                <artifactId>docker-maven-plugin</artifactId>
                <version>0.4.12</version>
                <configuration>
                    <!-- 注意imageName一定要是符合正则[a-z0-9-_.]的
，否则构建不会成功 -->
                    <!-- 详见：https://github.com/spotify/docker-
maven-plugin    Invalid repository name ... only [a-z0-9-_.] are
 allowed-->
                    <imageName>microservice-discovery-eureka-doc
kerfile</imageName>
                    <!-- 指定Dockerfile所在的路径 -->
                    <dockerDirectory>${project.basedir}/src/main
/docker</dockerDirectory>
                    <resources>
                        <resource>
                            <targetPath>/</targetPath>
                            <directory>${project.build.directory}
</directory>
                            <include>${project.build.finalName}.
jar</include>
                        </resource>
                    </resources>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

其他步骤一样。这样即可使用Dockerfile进行构建Docker镜像啦。

将Docker镜像push到DockerHub上

- 首先修改Maven的全局配置文件settings.xml，添加以下段落

```
<servers>
  <server>
    <id>docker-hub</id>
    <username>你的DockerHub用户名</username>
    <password>你的DockerHub密码</password>
    <configuration>
      <email>你的DockerHub邮箱</email>
    </configuration>
  </server>
</servers>
```

- 在DockerHub上创建repo,例如：test，如下图



- 项目pom.xml修改为如下：注意imageName的路径要和repo的路径一致

```
<build>
        <plugins>
            <!-- docker的maven插件，官网：https://github.com/spoti
fy/docker-maven-plugin -->
            <plugin>
                <groupId>com.spotify</groupId>
                <artifactId>docker-maven-plugin</artifactId>
                <version>0.4.12</version>
                <configuration>
                    <!-- 注意imageName一定要是符合正则[a-z0-9-_.]的
，否则构建不会成功 -->
                    <!-- 详见：https://github.com/spotify/docker-
maven-plugin Invalid repository
                        name ... only [a-z0-9-_.] are allowed -->

                    <!-- 如果要将docker镜像push到DockerHub上去的话，
这边的路径要和repo路径一致 -->
                    <imageName>eacdy/test</imageName>
                    <!-- 指定Dockerfile所在的路径 -->
                    <dockerDirectory>${project.basedir}/src/main
```

```
/docker</dockerDirectory>
                    <resources>
                        <resource>
                            <targetPath>/</targetPath>
                            <directory>${project.build.directory}
</directory>
                            <include>${project.build.finalName}.
jar</include>
                        </resource>
                    </resources>
                    <!-- 以下两行是为了docker push到DockerHub使用的
。 -->
                    <serverId>docker-hub</serverId>
                    <registryUrl>https://index.docker.io/v1/</re
gistryUrl>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

- 执行命令：

```
mvn clean package docker:build  -DpushImage
```

- 搞定，等构建成功后，我们会发现Docker镜像已经被push到DockerHub上了。

将镜像push到私有仓库

在很多场景下，我们需要将镜像push到私有仓库中去，这边为了讲解的全面性，私有仓库采用的是配置登录认证的私有仓库。

- 和push镜像到DockerHub中一样，我们首先需要修改Maven的全局配置文件settings.xml，添加以下段落

```
<servers>
  <server>
    <id>docker-registry</id>
    <username>你的DockerHub用户名</username>
    <password>你的DockerHub密码</password>
    <configuration>
      <email>你的DockerHub邮箱</email>
    </configuration>
  </server>
</servers>
```

- 将项目的pom.xml改成如下，

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.12</version>
  <configuration>
    <!-- 路径为：私有仓库地址/你想要的镜像路径 -->
    <imageName>reg.itmuch.com/test-pull-registry</imageName>
    <dockerDirectory>${project.basedir}/src/main/docker</dockerD
irectory>

    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>

    <!-- 与maven配置文件settings.xml一致 -->
    <serverId>docker-registry</serverId>
  </configuration>
</plugin>
```

- 执行：

```
mvn clean package docker:build  -DpushImage
```

稍等片刻，将会push成功。

- 如果想要从私服上下载该镜像，执行：

```
docker login reg.itmuch.com  # 然后输入账号和密码
docker pull reg.itmuch.com/test-pull-registry
```

将插件绑定在某个phase执行

在很多场景下，我们有这样的需求，例如执行 `mvn clean package` 时，自动地为我们构建docker镜像，可以吗？答案是肯定的。我们只需要将插件的 `goal` 绑定在某个phase即可。

所谓的phase和goal，可以这样理解：maven命令格式是： `mvn phase:goal` ，例如 `mvn package docker:build` 那么， `package` 和 `docker` 都是phase， `build` 则是goal 。

下面是示例：

首先配置属性：

```
<properties>
  <docker.image.prefix>reg.itmuch.com</docker.image.prefix>
</properties>
```

插件配置：

```
  <build>
    <plugins>
      <plugin>
        <groupId>com.spotify</groupId>
        <artifactId>docker-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>build-image</id>
            <phase>package</phase>
            <goals>
              <goal>build</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <imageName>${docker.image.prefix}/${project.artifactId}
</imageName>
          <baseImage>java</baseImage>
          <entryPoint>["java", "-jar", "/${project.build.finalNa
me}.jar"]</entryPoint>
          <resources>
            <resource>
              <targetPath>/</targetPath>
              <directory>${project.build.directory}</directory>
              <include>${project.build.finalName}.jar</include>
            </resource>
          </resources>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

如上，我们只需要添加：

```
        <executions>
          <execution>
            <id>build-image</id>
            <phase>package</phase>
            <goals>
              <goal>build</goal>
            </goals>
          </execution>
        </executions>
```

即可。本例指的是讲docker的build目标，绑定在package这个phase上。也就是说，用户只需要执行 `mvn package` ，就自动执行了 `mvn docker:build` 。

常见异常

连接不上2375（一般在Win7上出现）

```
Connect to localhost:2375 [localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] failed: Connection refused: connect -> [Help 1]
```

解决步骤：

- 输入 `docker-machine env`

```
$Env:DOCKER_TLS_VERIFY = "1"
$Env:DOCKER_HOST = "tcp://192.168.99.100:2376"
$Env:DOCKER_CERT_PATH = "C:\Users\Administrator\.docker\machine\machines\default
```

- 为插件添加配置

```
<!-- 解决Connect to localhost:2375的问题的其中一种方式，注意要跟docker
-machine env相一致 -->
<dockerHost>https://192.168.99.100:2376</dockerHost>
    <dockerCertPath>C:\Users\Administrator\.docker\machine\machi
nes\default</dockerCertPath>
```

修改后插件配置变为：

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.12</version>
    <configuration>
        <imageName>eacdy/test</imageName>
        <dockerDirectory>${project.basedir}/src/main/docker</doc
kerDirectory>

        <!-- 解决Connect to localhost:2375的问题的其中一种方式，注意
要跟docker-machine env相一致 -->
        <dockerHost>https://192.168.99.100:2376</dockerHost>
        <dockerCertPath>C:\Users\Administrator\.docker\machine\m
achines\default</dockerCertPath>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>

                <include>${project.build.finalName}.jar</include>

            </resource>
        </resources>
        <!-- 以下两行是为了docker push到DockerHub使用的。 -->
        <serverId>docker-hub</serverId>
        <registryUrl>https://index.docker.io/v1/</registryUrl>
    </configuration>
</plugin>
```

- 参考：https://github.com/spotify/docker-maven-plugin/issues/116

TIPS

1. imageName必须符合正则[a-z0-9-_.]，否则将会构建失败
2. 插件默认使用localhost:2375去连接Docker，如果你的Docker端口不是2375，需要配置环境变量 `DOCKER_HOST=tcp://<host>:2375`

代码地址（任选其一）

> https://git.oschina.net/itmuch/spring-cloud-study/tree/master/docker/microservice-discovery-eureka https://github.com/eacdy/spring-cloud-study/tree/master/docker/microservice-discovery-eureka

参考文档

> http://developer.51cto.com/art/201404/434879.htm https://linux.cn/article-6131-rss.html

3.8 Docker Compose

前文提到的Dockerfile 可以让用户管理一个单独的容器，那么如果我要管理多个容器呢，例如：我需要管理一个Web应用的同时还要加上其后端的数据库服务容器呢？Compose就是这样的一个工具。让我们看下官网对Compose的定义：

Compose 是一个用于定义和运行多容器的Docker应用的工具。使用Compose，你可以在一个配置文件（yaml格式）中配置你应用的服务，然后使用一个命令，即可创建并启动配置中引用的所有服务。下面我们进入Compose的实战吧。

我们使用最新的Docker Compose 1.8.0进行讲解。

3.8.1 Docker Compose的安装

安装Compose

Compose的安装有多种方式，例如通过shell安装、通过pip安装、以及将compose作为容器安装等等。本文讲解通过shell安装的方式。其他安装方式如有兴趣，可以查看Docker的官方文档：https://docs.docker.com/compose/install/

- 下载 `docker-compose` ，并放到 `/usr/local/bin/`

```
curl -L https://github.com/docker/compose/releases/download/1.8.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
```

- 为Docker Compose脚本添加执行权限

```
chmod +x /usr/local/bin/docker-compose
```

- 安装完成，测试：

```
docker-compose --version
```

结果显示：

```
docker-compose version 1.8.0, build f3628c7
```

说明Compose已经成功安装完成了。

安装Compose命令补全工具

按照上文讲解，我们已经成功地安装完Docker Compose。但是，我们输入 `docker-compose` 命令，按下TAB键，发现此时Compose并没有给我们该命令的提示，那么如何让命令给我们提示呢？我们需要安装Compose命令补全工具。

Compose命令补全在Bash和Zsh下的安装方式不同，由于笔者是使用CentOS 7进行讲解的，而CentOS 7默认使用Bash，故而本文只讲解命令补全在Bash下的安装，其他Shell以及其他系统上的安装，请查看Docker的官方文档：https://docs.docker.com/compose/completion/

```
curl -L https://raw.githubusercontent.com/docker/compose/$(docke
r-compose version --short)/contrib/completion/bash/docker-compos
e > /etc/bash_completion.d/docker-compose
```

这样，在重新登录后，输入 `docker-compose` 命令后，按下TAB键盘，效果如下：

```
[root@localhost ~]# docker-compose
build     config    down      exec      kill      pause     ps        p
ush       rm        scale     stop      up
bundle    create    events    help      logs      port      pull      r
estart    run       start     unpause   version
```

发现已经可以自动提示了。

3.8.2 Docker Compose入门示例

Compose的使用非常简单，只需要编写一个 `docker-compose.yml` ，然后使用 `docker-compose` 命令操作即可。 `docker-compose.yml` 描述了容器的配置，而 `docker-compose` 命令描述了对容器的操作。我们首先通过一个示例快速入门：

还记得前文，我们使用Dockerfile为项目 `microservice-discovery-eureka` 构建Docker镜像吗？我们还以此项目为例，在node0（192.168.11.143）这台机器上测试。

- 我们在 `microservice-discovery-eureka-0.0.1-SNAPSHOT.jar` 所在目录的上一级目录，创建 `docker-compose.yml` 文件。 目录树结构：

```
├── docker-compose.yml
└── eureka
    ├── Dockerfile
    └── microservice-discovery-eureka-0.0.1-SNAPSHOT.jar
```

- 然后在 `docker-compose.yml` 中添加内容如下：

```
eureka:
  build: ./eureka
  ports:
    - "8761:8761"
  expose:
    - 8761
```

- 在 `docker-compose.yml` 所在路径执行：

```
docker-compose up
```

发现打印日志：

```
eureka_1  | 2016-09-23 02:23:46.163  INFO 1 --- [           main
] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on p
ort(s): 8761 (http)
eureka_1  | 2016-09-23 02:23:46.164  INFO 1 --- [           main
] c.n.e.EurekaDiscoveryClientConfiguration : Updating port to 87
61
eureka_1  | 2016-09-23 02:23:46.167  INFO 1 --- [           main
] c.itmuch.cloud.study.EurekaApplication   : Started EurekaAppli
cation in 8.791 seconds (JVM running for 9.939)
eureka_1  | 2016-09-23 02:24:46.016  INFO 1 --- [a-EvictionTimer
] c.n.e.registry.AbstractInstanceRegistry  : Running the evict t
ask with compensationTime 0ms
```

- 访问： `http://宿主机IP:8761/` ，本文为： `http://192.168.11.143:8761/` ，发现可以正常启动。

3.8.3 docker-compose.yml常用命令

image

指定镜像名称或者镜像id，如果该镜像在本地不存在，Compose会尝试pull下来。

示例：

```
image: java
```

build

指定Dockerfile文件的路径。可以是一个路径，例如：

```
build: ./dir
```

也可以是一个对象，用以指定Dockerfile和参数，例如：

```
build:
  context: ./dir
  dockerfile: Dockerfile-alternate
  args:
    buildno: 1
```

command

覆盖容器启动后默认执行的命令。

示例：

```
command: bundle exec thin -p 3000
```

也可以是一个list，类似于Dockerfile总的CMD指令，格式如下：

```
command: [bundle, exec, thin, -p, 3000]
```

links

链接到其他服务中的容器。可以指定服务名称和链接的别名使用 `SERVICE:ALIAS` 的形式，或者只指定服务名称，示例：

```
web:
  links:
   - db
   - db:database
   - redis
```

external_links

表示链接到docker-compose.yml外部的容器，甚至并非Compose管理的容器，特别是对于那些提供共享容器或共同服务。格式跟links类似，示例：

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

ports

暴露端口信息。使用宿主端口:容器端口的格式，或者仅仅指定容器的端口（此时宿主机将会随机指定端口），类似于 `docker run -p` ，示例：

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
```

expose

暴露端口，只将端口暴露给连接的服务，而不暴露给宿主机，示例：

```
expose:
 - "3000"
 - "8000"
```

volumes

卷挂载路径设置。可以设置宿主机路径 （`HOST:CONTAINER`） 或加上访问模式 （`HOST:CONTAINER:ro`）。示例：

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro

  # Named volume
  - datavolume:/var/lib/mysql
```

volumes_from

从另一个服务或者容器挂载卷。可以指定只读或者可读写，如果访问模式没有指定，则默认是可读写。示例：

```
volumes_from:
 - service_name
 - service_name:ro
 - container:container_name
 - container:container_name:rw
```

environment

设置环境变量。可以使用数组或者字典两种方式。只有一个key的环境变量可以在运行Compose的机器上找到对应的值，这有助于加密的或者特殊主机的值。示例：

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

env_file

从文件中获取环境变量，可以为单独的文件路径或列表。如果通过 `docker-compose -f FILE` 指定了模板文件，则 `env_file` 中路径会基于模板文件路径。如果有变量名称与 `environment` 指令冲突，则以 `envirment` 为准。示例：

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

extends

继承另一个服务，基于已有的服务进行扩展。

net

设置网络模式。示例：

```
net: "bridge"
net: "host"
net: "none"
net: "container:[service name or container name/id]"
```

dns

配置dns服务器。可以是一个值，也可以是一个列表。示例：

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

dns_search

配置DNS的搜索域，可以是一个值，也可以是一个列表，示例：

```
dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com
```

其他

docker-compose.yml 还有很多其他命令，本文仅挑选常用命令进行讲解，其他不不作赘述。如果感兴趣的，可以参考docker-compose.yml文件官方文档：https://docs.docker.com/compose/compose-file/

参考文档

Docker Compose安装及使用： http://www.tuicool.com/articles/AnIVJn

Docker Compose使用全解：http://blog.csdn.net/zhiaini06/article/details/45287663

Docker Compose命令详解：http://blog.csdn.net/wanghailong041/article/details/52162293

3.8.4 docker-compose常用命令

TODO

参考：

Docker官方文档：https://docs.docker.com/compose/overview/

Dokcer教程：http://wiki.jikexueyuan.com/project/docker-technology-and-combat/install.html